AFIT/GCE/ENG/97D-01

# A COMPARATIVE ANALYSIS OF NETWORKS OF WORKSTATIONS AND MASSIVELY PARALLEL PROCESSORS FOR SIGNAL PROCESSING

THESIS

David C. Gindhart
First Lieutenant, USAF

AFIT/GCE/ENG/97D-01

19980121 066

Approved for public release; distribution unlimited

DTIC QUALITY INSPECTED 3

The views expressed in this thesis are those of the author and do not necessarily reflect the official policy or position of the Department of Defense or the U. S. Government.

AFIT/GCE/ENG/97D-01

# A COMPARATIVE ANALYSIS OF NETWORKS OF WORKSTATIONS AND MASSIVELY PARALLEL PROCESSORS FOR SIGNAL PROCESSING

THESIS

Presented to the Faculty of the Graduate School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science in Computer Engineering

David C. Gindhart, B.S.

First Lieutenant, USAF

December 1997

Approved for public release; distribution unlimited

AFIT/GCE/ENG/97D-01

# A COMPARATIVE ANALYSIS OF NETWORKS OF WORKSTATIONS AND MASSIVELY PARALLEL PROCESSORS FOR SIGNAL PROCESSING

THESIS

David C. Gindhart

Presented to the Faculty of the Graduate School of Engineering

of the Air Force Institute of Technology

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science in Computer Engineering

Dr. Gary Lamont
Member

Maj Richard Raines
Member

Lt Col David Gallagher
Chairman

## Acknowledgments

First and foremost, I would like to thank my family for their love and understanding. My wife, Julie, always supported me in so many different ways. Whether it was making sure life at home was in order or listening to my endless complaints, Julie was always there for me. I also want to thank my daughter Caroline who won't be able to read this for a few years, but always put things in perspective for me with a big hug and a "Hi Daddy" when I came in the door.

I would like to thank my thesis advisor, Lt Col David Gallagher. He is my mentor in many ways, as a professor, as an officer, and as a person. Despite his hectic schedule, he made time to talk and more importantly listen. I also appreciate the time and advice of my committee members, Dr. Gary Lamont and Maj Richard Raines.

Some of the most enjoyable times here have been with the great friends I've made. The list is too long to name them all, but they have given me memories I will never forget. Finally, and most importantly, I must thank God for delivering me to this point in my life. Every day I have prayed for His guidance and grace and He has bestowed more blessings on me than I can begin to count.

David C. Gindhart

# Table of Contents

Page

**Acknowledgments** ..... **ii**

**Table of Contents** ..... **iii**

**List of Figures** ..... **viii**

**List of Tables** ..... **xi**

**Abstract** ..... **xii**

**I. Introduction** ..... **1-1**

**II. Background** ..... **2-1**

2.1 Parallel Processing ..... 2-1

2.1.1 MPPs ..... 2-2

2.1.2 Advantages of MPPs ..... 2-2

2.1.3 Disadvantages of MPPs ..... 2-4

2.2 Networks of Workstations ..... 2-5

2.2.1 The Emergence of NOWs ..... 2-6

2.2.2 Challenges for NOWs ..... 2-7

2.2.3 Networks for NOWs ..... 2-7

2.2.4 Importance of Messaging Layers ..... 2-9

2.3 AFIT NOW ..... 2-11

2.3.1 Workstations ..... 2-11

2.3.2 Myrinet Network ..... 2-11

2.3.3 Messaging Layers ..... 2-13

2.3.4 MPI Communication Libraries ..... 2-16

2.4 MPPs Used in Research ..... 2-17
2.4.1 IBM SP2 ..... 2-17
2.4.2 Intel Paragon ..... 2-18
2.5 Related Research ..... 2-19
2.5.1 NOWs ..... 2-19
2.5.2 2-D FFT Research ..... 2-20

**III. Methodology ..... 3-1**
3.1 1-D FFT ..... 3-2
3.1.1 1-D FFT Theory ..... 3-2
3.1.2 1-D FFT Implementations ..... 3-6
3.1.2.1 FFTW ..... 3-6
3.1.2.2 Japanese Split-Radix FFT ..... 3-8
3.2 2-D FFT ..... 3-8
3.2.1 2-D FFT Algorithms ..... 3-9
3.2.1.1 Row-Column Algorithm ..... 3-9
3.2.1.2 Vector-Radix Algorithm ..... 3-10
3.2.2 2-D FFT Implementations ..... 3-12
3.2.2.1 Row-Column Implementation ..... 3-13
3.2.2.2 Vector-Radix Implementation ..... 3-15
3.2.2.3 Pipeline Implementation ..... 3-18
3.2.3 Theoretical Complexity Analysis ..... 3-21
3.2.3.1 Isoefficiency Analysis ..... 3-21
3.2.3.2 Computational Complexity ..... 3-23
3.2.3.3 Communication Complexity ..... 3-24
3.2.3.4 Pipeline Complexity ..... 3-26

3.2.3.5 Complexity Summary ..... 3-27

3.2.4 2-D FFT Code Optimizations ..... 3-28

3.2.4.1 Loop Invariant Code Removal and Common Sub-expression Elimination ..... 3-28

3.2.4.2 Derived Datatypes ..... 3-29

3.2.4.3 Packing ..... 3-30

3.2.4.4 Non-blocking (Asynchronous) Communication ..... 3-30

3.2.4.5 Row-Column Optimizations ..... 3-31

3.2.4.6 Vector-Radix Optimizations ..... 3-32

3.2.4.7 Pipeline Optimizations ..... 3-32

3.3 Experimental Framework ..... 3-33

3.3.1 Input Size ..... 3-33

3.3.2 Number of Processors ..... 3-34

3.3.3 Messaging Layers ..... 3-34

3.3.3.1 AFIT NOW ..... 3-34

3.3.3.2 SP2 ..... 3-35

3.3.3.3 Paragon ..... 3-36

3.3.4 Compilers and Compiler Optimizations ..... 3-36

3.3.5 MPI Implementations ..... 3-37

3.3.5.1 AFIT NOW ..... 3-37

3.3.5.2 MPPs ..... 3-38

3.3.6 Number of Iterations/Runs ..... 3-39

**IV. Analysis ..... 4-1**

4.1 Benchmark Results and Metric Comparisons ..... 4-2

4.1.1 Computation Benchmark ..... 4-2

4.1.2 Communication Benchmark ..... 4-5
4.1.3 Metrics ..... 4-10
4.1.4 Scalability ..... 4-12
4.2 Paragon Results ..... 4-13
4.2.1 Vector-Radix ..... 4-13
4.2.1.1 Optimizations ..... 4-15
4.2.1.2 Scalability ..... 4-16
4.2.2 Row-Column ..... 4-17
4.2.2.1 Optimizations ..... 4-19
4.2.2.2 Scalability ..... 4-20
4.2.3 Pipeline ..... 4-21
4.2.3.1 Optimizations ..... 4-22
4.2.3.2 Scalability ..... 4-22
4.2.4 Row-Column vs. Vector-Radix vs. Pipeline ..... 4-23
4.3 SP2 Results ..... 4-26
4.3.1 Vector-Radix ..... 4-27
4.3.1.1 Optimizations ..... 4-28
4.3.1.2 Scalability ..... 4-29
4.3.2 Row-Column ..... 4-30
4.3.2.1 Optimizations ..... 4-32
4.3.2.2 Scalability ..... 4-33
4.3.3 Pipeline ..... 4-35
4.3.4 Row-Column vs. Vector-Radix vs. Pipeline ..... 4-37
4.4 AFIT NOW ..... 4-39
4.4.1 Effect of Network and Messaging Layers on Performance ..... 4-39

4.4.2 Vector-Radix ..... 4-40

4.4.3 Row-Column ..... 4-42

4.4.4 Pipeline ..... 4-43

4.4.5 Vector-Radix vs. Row-Column vs. Pipeline ..... 4-45

4.5 Paragon vs. SP2 vs. AFIT NOW ..... 4-47

4.5.1 Vector-Radix ..... 4-48

4.5.2 Row-Column ..... 4-48

4.5.3 Pipeline ..... 4-49

4.6 Best Algorithm ..... 4-50

4.7 Best Platform ..... 4-51

**V. Conclusion ..... 5-1**

5.1 Conclusions ..... 5-1

5.1.1 Parallel 2-D FFT Performance ..... 5-1

5.1.2 2-D FFT Algorithms ..... 5-3

5.1.3 Computational Platforms ..... 5-7

5.2 Contributions ..... 5-9

5.3 Recommendations for Future Work ..... 5-11

5.4 Future of Networks of Workstations ..... 5-12

**Bibliography**

**VITA**

# List of Figures

Page

Figure 3-1. 8 Point Radix-2 DIT FFT. [ODonnell96] ..... 3-4

Figure 3-2. 8 Point Radix-2 DIF FFT. [ODonnell96] ..... 3-4

Figure 3-3. Number of Multiplication and Additions for 1-D FFT Algorithms. ..... 3-6

Figure 3-4. Vector-Radix Butterfly. [Pitas93] ..... 3-11

Figure 3-5. Matrix Access of 8x8 Vector-Radix 2-D FFT. ..... 3-12

Figure 3-6. Scatter of 8x8 Image in Row-Column 2-D FFT. ..... 3-13

Figure 3-7. Matrix Transposition Steps of Row-Column 2-D FFT. ..... 3-14

Figure 3-8. Gather Step of Row-Column 2-D FFT ..... 3-15

Figure 3-9. Bit Reverse Step of Vector-Radix 2-D FFT ..... 3-15

Figure 3-10. Communication During Vector-Radix 2-D FFT with 8 Processors ..... 3-16

Figure 3-11. Matrix Access Pattern During Communication Stage of Vector-Radix 2-D FFT. ..... 3-17

Figure 3-12. Gather and Permutation Step of Vector-Radix 2-D FFT. ..... 3-18

Figure 3-13. Overview of Pipeline 2-D FFT. ..... 3-20

Figure 3-14. Source/Destination Processor Sending to Row Processors ..... 3-20

Figure 3-15. Row Processors Sending to Column Processors. ..... 3-21

Figure 3-16. Column Processors Send to Source/Destination Processor. ..... 3-21

Figure 3-17. Number of Messages Sent for the Three Implementations on a Logarithmic Scale ..... 3-25

Figure 3-18. Megabytes Sent by the Three Implementations. ..... 3-25

Figure 3-19. Timing of Pipeline Implementation. ..... 3-27

Figure 3-20. Loop Invariant Code Removal and Common Sub-expression Elimination ..... 3-29

Figure 4-1. Average Megaflops of Four Serial 2-D FFTs on a 200 MHz Ultra Sparc. ... 4-3

Figure 4-2. Average Megaflops of Serial Row-Column 2-D FFT for Each Platform ... 4-4

Figure 4-3. Message Latency for Each Platform for Small Message Sizes. ..... 4-6

Figure 4-4. Measured Throughput of Each Platform for Large Message Sizes..... 4-8

Figure 4-5. Time Breakdown of Vector-Radix with 4 Processors on the Paragon..... 4-14

Figure 4-6. Time Breakdown of Vector-Radix for 512x512 Input Size on the Paragon.4-15

Figure 4-7. Speedup of Vector-Radix on the Paragon ..... 4-17

Figure 4-8. Time Breakdown of Row-Column with 4 Processors on the Paragon..... 4-18

Figure 4-9. Time Breakdown of the Row-Column for 512x512 Input Size on the Paragon. ..... 4-18

Figure 4-10. Percent Change in Runtime as N/p Varies for AlltoAll Optimizations in Row-Column on the Paragon..... 4-19

Figure 4-11. Speedup of Row-Column on the Paragon..... 4-21

Figure 4-12. Speedup of Pipeline on the Paragon..... 4-23

Figure 4-13. Speedup of the Three Algorithms for 1024x1024 Input Size on the Paragon.4-24

Figure 4-14. Efficiency of the Three Algorithms for 1024x1024 Input Size on the Paragon. ..... 4-25

Figure 4-15. Time Breakdown of Vector-Radix with 4 Processors on the SP2. ..... 4-27

Figure 4-16. Time Breakdown of Vector-Radix for 512x512 Input Size on the SP2. . 4-28

Figure 4-17. Speedup of Vector-Radix on the SP2. ..... 4-30

Figure 4-18. Time Breakdown of Row-Column with 4 Processors on the SP2 ..... 4-31

Figure 4-19. Time Breakdown of Row-Column for 512x512 Input Size on the SP2. . 4-31

Figure 4-20. Speedup of Row-Column on the SP2. ..... 4-34

Figure 4-21. Speedup of the Pipeline on the SP2. ..... 4-37

Figure 4-22. Speedup of the Three Algorithms for 1024x1024 Input Size on the SP2.4-38

Figure 4-23. Efficiency of the Three Algorithms for 1024x1024 Input Size on the SP2.4-38

Figure 4-24. Time Breakdown of Vector-Radix with 2 and 4 Processors on the AFIT NOW..... 4-41

Figure 4-25. Speedup of Vector-Radix on the AFIT NOW..... 4-41

Figure 4-26. Time Breakdown of Row-Column with 2 and 4 Processors on the AFIT NOW..... 4-42

Figure 4-27. Speedup of Row-Column on the AFIT NOW..... 4-43

Figure 4-28. Speedup of Pipeline on the AFIT NOW. ..... 4-44

Figure 4-29. Time Breakdown of Pipeline with 3 and 5 Processors on the AFIT NOW.4-44

Figure 4-30. Speedup of the Three Algorithms with 2 and 4 Processors on the AFIT NOW..... 4-45

Figure 4-31. Efficiency of the Three Algorithms with 2 and 4 Processors on the AFIT NOW..... 4-46

Figure 4-32. Performance in Megaflops of Row-Column with 2 and 4 Processors on the Three Platforms. ..... 4-49

Figure 4-33. Performance in Megaflops of the Pipeline with 3 and 5 Processors on the Three Platforms. ..... 4-50

Figure 4-34. Performance in Megaflops of the Best Algorithm on Each Platform with 2/3 and 4/5 Processors. ..... 4-52

# List of Tables

Page

Table 2-1. Latency and Throughput of Messaging Layers for Myrinet. ..... 2-16

Table 3-1. Number of Operations for Row-Column and Vector-Radix Implementations.3-24

Table 4-1. 0-Byte Latency and Message Startup Time for Each Platform. ..... 4-7

Table 4-2. Maximum Theoretical Bandwidth and Measured Throughput of Each Platform. ..... 4-9

Table 4-3. Message Startup Time and Message Transfer Time for the Three Platforms on a Scatter or Gather Operation with 4 Processors and Varying Input Sizes..... 4-10

Table 4-4. Percentage Decrease in Runtime of Vector-Radix Optimizations on the Paragon. ..... 4-16

Table 4-5. Percentage Change in Runtime for Gather Optimization in Row-Column Code on the Paragon. ..... 4-20

Table 4-6. Percentage Change in Runtime of Optimizations of Vector-Radix on the SP2.4-29

Table 4-7. Percentage Change in Runtime of Unoptimized Row-Column on the SP2. 4-32

Table 4-8. Ratio of FFT Computation Time to Matrix Transpose Time in Serial Row-Column 2-D FFT on the SP2. ..... 4-35

Table 4-9. Performance in Megaflops of the Three Algorithms with Different Network and Messaging Layers on the AFIT NOW. ..... 4-40

Table 4-10. Average Decrease in Runtime of Pipeline vs. Row-Column and Vector-Radix for the Three Platforms. ..... 4-50

Table 4-11. Runtime of Paragon Pipeline with 9 processors and Serial 2-D FFT on 200 MHz Ultra. ..... 4-53

## Abstract

The traditional approach to parallel processing has been to use Massively Parallel Processors (MPPs). An alternative design is commercial-off-the-shelf (COTS) workstations connected to high-speed networks. These networks of workstations (NOWs) typically have faster processors, heterogeneous environments, and most importantly, offer a lower per node cost.

This thesis compares the performance of MPPs and NOWs for the two-dimensional fast Fourier transform (2-D FFT). Three original, high-performance, portable 2-D FFTs have been implemented: the vector-radix, row-column and pipeline. The performance of these algorithms was measured on the Intel Paragon, IBM SP2 and the AFIT NOW, which consists of 6 Sun Ultra workstations connected via the Myrinet switch.

Three important conclusions have been made. First, the pipeline was the best algorithm on all platforms by approximately 30%. Second, the NOW was nearly equal to the SP2 in runtime, while the Paragon did not outperform a single Ultra workstation. As a result, NOWs are a competitive platform for this application. Finally, only limited speedup was achieved on the SP2 (2.9) with 32 processors, and AFIT NOW (1.9) with 5 processors. It appears that the changing platform communication-to-computation ratio has made the 2-D FFT a less viable candidate for parallelization, given its high communication overhead.

## *I. Introduction*

As today's computing power has scaled, so have the problems that need to be solved by it. Some of these problems are summarized in the Grand Challenge problems put forth by the U.S. Government.[HPCCITS96] Grand Challenge problems are scientific and engineering applications whose solutions have economic and scientific impact on our country. The solutions to these problems are generally estimated through simulation because experiments are not able to be accomplished given the size, speed, distance, health, safety or economic factors in the application domain. Some examples of Grand Challenge problems are computational fluid dynamics, quantum dynamics, atomic construction and decay, and black hole interaction. Beyond these computationally difficult problems, there are problems in the NP class which no amount of conventional computing power can solve in a reasonable amount of time. As the single processor model that was developed by Von Neumann reached its limitations, computer architects moved towards exploiting parallelism in problem solutions to achieve higher performance. Parallelization can be found in uniprocessor architectures that employ pipelining, superscalar, and VLIW techniques. These techniques, however, have not provided the additional computing resources necessary to solve all computationally difficult problems. As a result, computer architects developed multiple processor systems to further exploit the available parallelism.

Multiple processor systems have taken many forms, as designers have attempted to achieve the maximum performance with the current technology. Two primary designs have emerged which differ primarily in their communication mechanism: shared memory

and message passing.[Hwang90] Shared memory machines communicate by reading and writing data to a globally visible memory. Message passing machines communicate via the sending and receiving of data in messages. Message passing machines generally use many more processors compared to shared memory machines and are often termed *massively parallel processors* (MPPs). These two multi-processor designs are the basis for most parallel platforms today.

MPPs are a popular choice for parallel processing, but they do have some disadvantages. MPPs are typically designed around a commodity microprocessor, with a high speed communication network and other supporting hardware. Because of their complexity, these machines may take two to three years of research and development. Also, operating system and communication library software must be developed for the parallel platform, adding to their development time. This design time and engineering effort generally results in fast communication speeds, but with an older, less state-of-the-art microprocessor. The biggest disadvantage of MPPs, however, may be their extremely high cost, which is further exacerbated by their small volume of sales.

To overcome some of the disadvantages of MPPs, computer architects are exploring the idea of using low-cost, commercial off-the-shelf (COTS) workstations, connected to a COTS network, to try to achieve the performance of MPPs. Three major factors have recently made networks of workstations (NOWs) an effective alternative to MPPs for parallel processing.[Anderson95] First, the volume of workstations sold has skyrocketed in the past five years. This increased volume allows manufacturers to

amortize design costs across many more units, which drives per workstation cost down. Second, technological advances and competition for a piece of the widening workstation market has spurred performance advances at an ever-increasing pace. As a result, vendors are putting better and better workstations on the desktop every few months. Third, high bandwidth, low latency, COTS networks have recently become available to use as NOW interconnection networks. These networks provide orders of magnitude increase in maximum, hardware bandwidth over traditional NOW networks. These three factors combined have greatly improved the effectiveness of NOWs, but there are still some obstacles in their path towards displacing MPPs.

There are two main challenges that NOW designers face in unlocking the potential of NOWs.[Anderson95] First, without a global operating system to coordinate resources, efficiency is hindered due to a lack of scheduling. Second, the observed network throughput of NOWs is far below their maximum hardware bandwidth, partly because of inefficient messaging layers. Recent research in improved messaging layers have reduced this limitation and new operating systems are being developed. However, until NOWs can communicate near the speed of MPPs with efficient messaging layers and coordinate their resources with a global operating system, they will still lack the performance of MPPs, despite their recent gains.

To answer the general question of whether NOWs are comparable to MPPs in performance, an extensive benchmark comparison would have to be made on a variety of applications. Of course, the choice of benchmarks and the latitude given to vendors for

performance tuning could have major effects on the results. This thesis effort does not attempt to make such a comparison for many reasons, not the least of which is the time constraints of this research. Instead, a particular application was chosen which is an important Air Force research area, but also is representative of a larger class of problems.

The application chosen as the comparison tool between MPPs and NOWs is the two-dimensional Fast Fourier Transform (2-D FFT). This application is part of the more general signal and image processing computational technology area, which is of great importance to the DoD. In fact, this research is supported by the Air Force Laboratory at Rome, NY under the umbrella of the High Performance Computing Modernization Program. This research is also part of a larger AFIT ongoing research project in computational signal and image processing. Signal and image processing is an integral part of such combat functions as intelligence gathering, target recognition and electronic warfare.

Another aspect of the 2-D FFT which makes it attractive as a comparison tool is its extensibility to other problem domains. The 2-D FFT is representative of many applications which have a regular matrix data structure and well-defined communication patterns. Many applications, not solely in signal and image processing, follow the pattern of distributing data equally to all processors, performing computations, exchanging data, and finally collecting the data to a single destination. Because of its extensibility and applicability, the 2-D FFT was chosen as the single comparison tool for this research. Given an application for comparison and a framework with which to make the comparison, the thesis objective statement becomes:

***Can NOWs meet or exceed the performance of MPPs for a particular signal and image processing application?***

To answer this question, two primary objectives must be met. The first objective is the development of parallel 2-D FFT implementations. This objective meets the sponsor's requirement for high performance, scaleable parallel software for signal and image processing. Towards this end, many possible optimizations should be investigated to obtain the best possible code. Once the optimized code has been finalized, the effectiveness of the optimizations can be analyzed. From the optimized code, performance results should indicate the best algorithm for a particular platform, input size and number of processors. The performance of the 2-D FFT implementations then become the basis of the comparison between MPPs and NOWs.

The second objective is to determine whether NOWs can offer the performance of MPPs for this application. Using, the best results from each platform, the platforms can be compared for performance and scalability. From this data, some conclusions can be made concerning the suitability of MPPs and NOWs for signal and image processing. The performance data should also indicate whether NOWs can be competitive with MPPs for this application.

The remainder of this document is organized as follows:

**Chapter II** gives the background necessary to understand the differences between MPPs and NOWs. The advantages and disadvantages of both types of platforms are discussed. The changes that have brought about the emergence of NOWs are described,

as well as the challenges NOWs still face. Finally, a detailed description of the NOW and MPPs used in this research is provided.

**Chapter III** has two primary purposes: description of both the comparison tool and the comparative methodology. In the first part of Chapter III, the 2-D FFT is explained and two common algorithms for its calculation are described. Three implementations are detailed , as well as the numerous optimizations used to improve the performance of the code. The second part of the chapter outlines the framework of the comparison. This description is given to facilitate the repeatability of the results, as well as to highlight the platform differences which may effect the comparison.

**Chapter IV** reports and analyzes the results of the 2-D FFT performance on the three platforms. First, the results from each algorithm are analyzed on each platform. Second, the three implementations are compared on each platform to determine the best code(s) for that platform. Third, the algorithms are compared across platforms to determine the best platform for a given algorithm. Finally, the best performance results from each platform are compared to analyze the performance of MPPs and NOWs for the 2-D FFT.

**Chapter V** presents conclusions about the suitability of NOWs and MPPs for image and signal processing, as well as a quantitative and qualitative answer to the question: Can NOWs perform comparably to MPPs for a particular signal and image processing application?

# *II. Background*

The purpose of this chapter is to give background information on MPPs and NOWs, in order to understand the performance differences between them. By studying the underlying platforms, a better understanding of the performance data collected and analyzed in Chapter IV can be gained. In the first part of this chapter, MPPs and NOWs are described, including the advantages and disadvantages of both. Next, the recent emergence of NOWs and the keys to their performance are discussed. Finally, the AFIT NOW is detailed, with a short discussion of two MPPs used in this research.

## *2.1 Parallel Processing*

The traditional approach to parallel processing was to use dedicated, custom-made multiprocessors. Initially, many of these were supercomputers with extremely large numbers of processors (>256) that executed the same instruction on different data sets. Under Flynn's classification these would be SIMD (Single Instruction, Multiple Data) architectures.[Flynn72] The extreme of the SIMD architecture was the CM-2 (1990) which had up to 65,536 processors, operating on one bit of data at a time.[Hwang90] A more recent trend has been towards the MIMD (Multiple Instruction, Multiple Data) architecture in which multiple processors operate independently with different program counters on separate data sets in an asynchronous manner. Within the MIMD model, there are two main paradigms for data sharing: *shared memory* and *message passing*. In the first model, shared memory, processors communicate by accessing a global address space through either a global memory or another processor's local memory. Careful attention

must be paid to synchronizing access to memory locations, known as *memory consistency*. Because of this problem, shared memory machines are typically limited to smaller numbers of processors than message passing computers. This research does not use shared memory machines in order to limit the scope of the problem.

### *2.1.1 MPPs*

In the second model, message passing, processors communicate by sending explicit messages and there is no global address space. Typically, message passing machines have larger numbers of processors and are referred to as Massively Parallel Processors (MPPs). MPPs have become more popular because of their hardware scalability. In the shared memory architectures, the interconnection network (IN) from the processors to the global memories carries all memory to processor traffic. As the number of processors increase, the IN becomes a bottleneck.[Hwang90] This is not true however in message passing machines where the number of processors can increase into the hundreds. Because of their scalability and the availability of these machines for our use, this thesis concentrates on MPPs when discussing traditional parallel processing.

### *2.1.2 Advantages of MPPs*

There are two main advantages to the traditional MPP approach. The first advantage is the communication performance available to the processors in a MPP. There are several architectural reasons for this good communication performance. The first advantage is that the network interface card (NIC) may be connected to the memory

bus, putting it close to the processor.[Anderson95] The advantage of this scheme is that memory buses have high bandwidth and low latency, especially when compared to I/O buses. I/O buses generally have to follow open standards and have to connect a variety of devices.[Hennessy96] An example of the performance difference between a memory and I/O bus can be seen in Section 2.3, where the Sun Ultra workstation is detailed. The second reason for good communication performance in MPPs is the presence of a fast, dedicated network processor to handle network traffic, connected via a wide, fast bus. For example, in the Intel Paragon there is an Intel i860 processor used for computations and an i860 used for network processing. These processors communicate via the fast and cache coherent memory which greatly improves performance over the I/O bus.[Hennessy96]

A second advantage of MPPs is that programmers are presented a single, global machine on which to run their program.[Anderson95] When a user starts a parallel program, he is granted all the available processors (if needed) and the program runs without competition from other users' processes. If other users are simultaneously trying to run other parallel jobs or running local processes, one parallel process may be active while another is not because it has been context-switched out. Without global scheduling, there is no guarantee that the parallel processes are running concurrently. This lack of scheduling causes processes to be waiting for data from a process that isn't even running, which results in unnecessary delay. Also, the user does not have to start the processes interactively because the spawned processes are started and controlled by the global operating system.

### *2.1.3 Disadvantages of MPPs*

The advantages of MPPs come at a high monetary cost and this high cost comes for several reasons. First, although today's MPP manufacturers attempt to use commodity parts (e.g. Intel, IBM or DEC microprocessors), there is still proprietary hardware that is used to create an MPP. This hardware is expensive, and more importantly, requires years of engineering and development. This development process many include repackaging of chips and re-routing of buses.[Anderson95] A second reason for high cost in MPPs is the operating system and software cost. Each new version of an MPP may be different enough from the last to cause a rewrite of the operating system and any other supporting software such as communication libraries. Lately MPP manufacturers have begun loading the full UNIX operating system on each node which decreases the operating system cost. However there must still be an overarching, global operating system which controls the processors as a whole for scheduling. Another, more hidden, cost of MPPs is the high cost of annual maintenance of the machine.

Ignoring the high cost of MPPs, there is a performance disadvantage in using MPPs: computational power. MPP manufacturers must design their system around a commodity microprocessor that is or will soon be commercially available. Once the microprocessor is chosen, the design, engineering and manufacturing process may take several years. As a result, the MPP does not have the highest performance microprocessor available by the time it hits the market. Since microprocessors improve their performance by 50% to 100% per year, the MPP will employ processors that may

have, at the extreme, 2 to 4 times less computational power per processor than currently available workstations.[Hennessy96, Anderson95]

In recent years, MPP manufacturers have tried to overcome these disadvantages by using more commodity parts. A good examples is the IBM SP2, which is one of the few MPPs to survive in recent years.[IBM97] The SP2 uses the RS/6000 line of microprocessors which allows them to use the same supporting hardware and software for each new processor in the series. It also allows the use of heterogeneous processors because of their backward compatibility. Because the NIC connects to the I/O bus, not the memory bus, changes at the processor/memory hierarchy level have little effect on the network controller design. With the existing SP2 infrastructure, a newer, faster processor can be inserted without causing a complete redesign of the system. Also, since the processor interfaces remain the same, newer supporting hardware and software can be developed without complete re-engineering. This SP2 design scheme has made it one of the premier MPPs available today.

### *2.2 Networks of Workstations*

With these problems of MPPs, there is an emerging alternative that overcomes many of the disadvantages of MPPs: a network of workstations (NOW). A workstation is a general term for any high-end PC (usually running Windows NT or Linux) or UNIX-based computer that has an operating system, I/O capability and a network connection.[Anderson95]

### *2.2.1 The Emergence of NOWs*

There are two recent trends in computer technology that have given NOWs advantages that were previously not available.[Anderson95] The first trend is the increase in microprocessor performance. This trend was discussed in Section 2.1 as a disadvantage of MPPs. Since the development of workstations takes less time, manufacturers can base their design on a newer microprocessor, thus yielding higher raw computational performance. Another advantage of workstations is that they can take advantage of low price commodity RAM and increasing disk drive sizes to offer larger aggregate memory and disk capacity than MPPs. MPPs typically have to use platform specific RAM and I/O devices.

The second trend in technology that benefits NOWs is volume. Workstations are cheaper than MPPs and sold in much higher quantity. For example, from the years 1989 to 1994, the sales ratio of PCs to supercomputers was 30,000:1.[Anderson95] This ratio has only grown since 1994 as PC sales have skyrocketed. The increase in volume of sales of workstations allows vendors to amortize design and engineering costs across many more machines and lower the per unit price as compared to MPPs. There are also other volume-related factors that improve per unit costs. A formula put forth by Gordon Bell, a computer engineer at Digital Equipment Corporation, states that doubling volume reduces unit cost by 90%.[Anderson95]

### *2.2.2 Challenges for NOWs*

It appears that NOWs can overcome the disadvantages of MPPs, mainly high cost and computational performance. The question is whether NOWs can equal the advantages of MPPs, mainly global operating systems and good communication performance.

There are several operating system opportunities for NOWs that can provide both parallel programs and everyday users with better performance. For one, if the combined memory of workstations can be made available to all the workstations, then processes can avoid swapping to disk by accessing other workstations' RAM through the network. This is beneficial because the time to access the local disk may be orders of magnitude greater than the time to access the RAM of another workstation, even with network delays. Another opportunity for NOWs it to provide the disk drives of all workstations globally in a RAID scheme. This scheme provides greater disk size, faster disk I/O and fault tolerance by distributing disk accesses across the workstations. The basis for all these opportunities must be a global operating system to coordinate resource sharing and a fast network to provide the resources quick enough to decrease the access time versus solely local disk access.[Anderson95]

### *2.2.3 Networks for NOWs*

Communication networks for NOWs are the hardware key to unleashing the computational power of NOWs. The first network used for NOWs was the 10 megabits

per second (Mbps) bandwidth Ethernet. This network provides a cheap, shared medium on which workstations compete for the use of the communication channel. As traffic increases, nodes collide as they try to simultaneously use the network. When collisions occur, the nodes must wait a minimum, random amount of time before they attempt to re-access the network. Each time they collide, the minimum time for re-access increases exponentially and as a result performance degrades quickly. An alternative to a shared medium such as Ethernet is a switched medium in which multiple sets of nodes can communicate in parallel. This scheme requires more complex hardware, namely a switch, but can increase performance greatly over a shared medium. Along with switching, there are higher speed Ethernet implementations emerging such as 100 Mbps and into the Gbps range. An alternative to Ethernet is Asynchronous Transfer Mode (ATM) which is a switched network that currently provides 155 Mbps bandwidth. ATM's primary focus has been to offer a variety of services including voice, data and video at high speeds for general Local Area Network (LAN) use, but not to support parallel processing. Finally, a third emerging NOW network is the Myrinet switch which operates at 1.28 Gbps.[Boden95] Myrinet is essentially a crossbar switch taken from a MPP and repackaged for NOWs. Reduction in VLSI feature size has allowed manufacturers to shrink and package a single crossbar switch as a high speed, yet affordable network. The advantage of a crossbar switch is its non-blocking topology which increases total switch throughput. The disadvantage of a crossbar is its high cost and lack of scalability. The number of links in a crossbar is $O(n^2)$ and so the increase in complexity, and therefore cost, scales with the square of the number of inputs.

### *2.2.4 Importance of Messaging Layers*

As the speed of these interconnection networks has increased, there has not necessarily been an equal increase in performance seen by applications. This performance lag occurs because the messaging layers of the seven layer OSI model, through which an application must communicate to the network hardware, represent significant overhead compared to the low latency of today's networks. The messaging layers provide the following functions: message delivery, message ordering, deadlock prevention, overflow prevention and error-free delivery. Message delivery is simply the transferring of data from one process to another. Message ordering and error-free delivery provide the message to the receiver in exactly the same way as the sender sent it. Deadlock and overflow prevention is provided so the programmer doesn't have to worry about buffer allocation and management.[Karamcheti94]

The messaging layer used for a NOW is an important factor in network performance and hence parallel program performance. The primary messaging layer used for day to day internetworking in many LANs is the Transport Control Protocol/Internet Protocol (TCP/IP). TCP/IP provides a platform-independent and vendor-independent protocol which allows heterogeneous computers to communicate across a desktop or across the world. TCP/IP was developed in the 1970s under the assumption that the underlying network hardware was not reliable and not very fast (compared to today's networks). As a result of this assumption, there is a great deal of software overhead in ensuring the guarantees of a messaging layer such as error-free, in-order

delivery.[Hennessy96] Another disadvantage of current TCP/IP implementations is that in order to do the communication processing, the processor must context switch from user mode into supervisor mode. This context switch stalls the pipeline and usually requires the cache to be flushed.[Culler96] These delays are orders of magnitude greater than the transmission time for a gigabit network. Because of these limitations, NOW developers have looked to faster and more efficient messaging layers which combine the functionality of multiple layers and eliminate unnecessary guarantees.

To be fair, there are other factors within the hardware that affect messaging layer performance beyond the messaging layer software. The first factor is the distance of the NIC from the processor. If the NIC is connected to the memory bus, which is usually cache coherent, higher performance is achieved. If the NIC is attached to the I/O bus, there is significant performance loss. The second factor is the processor that is used on the NIC. A faster processor on the NIC can increase DMA speed to the network and to the processor. For example, the Intel Paragon uses a 50 MHz i860 RISC processor for communication, the IBM SP2 uses a Power PC 601 processor, while the Myrinet Lanai contains a slower 20 MHz CISC processor.[Intel95, IBM97, Boden95] The third factor in messaging speed is the tradeoff between programmed I/O and DMA for message transfer between main memory and the NIC. Programmed I/O, load and store instructions, provide lower latency because the data is immediately transferred without having to wait until the DMA is setup. DMA provides better throughput since it is generally faster to transfer large blocks of data, but the latency increases because the DMA transfer must be coordinated between the NIC and processor.

## *2.3 AFIT NOW*

The AFIT NOW consists of six Sun workstations connected via the high-speed Myrinet switch. The AFIT NOW has been in existence in its current form since October 1996.

### *2.3.1 Workstations*

The workstations used for the NOW in this project are Sun Ultra Sparc Models 170 and 200. The heart of the Ultra is a 170 or 200 MHz, four-way superscalar Sparc Version 9 processor. There are two integer arithmetic logic units (ALUs) and 2 Floating Point (FP) ALUs. The FP functional units are pipelined with a FP add and multiply both taking 3 cycles. There is a 32 KB, on-chip, level 1 cache which is made up of a 16 KB direct-mapped data cache and a 16 KB 2-way set-associative instruction cache. There is also a 512 KB level 2 cache. Each Ultra has 128 MB of RAM and two 1 GB local hard drives. The I/O bus, known as the SBus, operates at 25 MHz and has a 64 bit wide datapath.[Sun97]

### *2.3.2 Myrinet Network*

The network in the AFIT NOW is an 8x8 Myrinet crossbar switch. Each link within the Myrinet switch provides 1.28 gigabits per second (Gbps) in each direction, yielding 2.56 Gbps total bandwidth in full duplex mode. Another important aspect of the Myrinet switch is its extremely low error rate. This reliability comes from the short distances within the switch and short cables or fiber optic links used for interconnection.

This type of network with its short distances is called by Myrinet a System Area Network (SAN). The result of low error rates is that the messaging layers can be constructed to perform less error control (error detection, retransmission, etc.) than those based on unreliable LAN or WAN links. The ATOMIC project which was the basis for the Myrinet switch experienced no bit errors or dropped packets while transferring in excess of $10^{15}$ bits.[Boden95]

Another important part of understanding the Myrinet communication system is the delivery of a message from application to application through the network. For a message to be delivered from the processor to the network, it must first go through the memory hierarchy, cache and main memory, on the memory bus (MBus). Since the network interface is connected to the I/O bus (SBus), the message must next traverse the SBus. The MBus (83 MHz) offers better performance than the SBus (25 MHz), so the SBus becomes a bottleneck during the message transfer. The SBus has a transfer rate of 23.9 MB/s for programmed I/O. When DMA is used for transfer, however, it can peak at 40-54 MB/s when large amounts of data are transferred.[Pakin95] The tradeoff is between latency and throughput as described earlier in section 2.2.4. Next, the message reaches the Myrinet NIC known as the Lanai. The Lanai is composed of a CISC processor that operates at the SBus frequency and achieves approximately 5 MIPS. The Lanai has 128K of SRAM which stores the Myrinet Control Program (MCP) and buffers for incoming and outgoing packets.[Boden95] It is important to note that the MCP is Myricom's implementation of the control program and that other groups have implemented different control programs for their messaging layers.

The memory of the Lanai is mapped into the address space of the host. As a result, the host can read and write to the Lanai memory through programmed I/O. The host cannot initiate a DMA transfer, but the Lanai must interrupt the host CPU to perform the DMA. All DMA transfers must be to or from a pinned-down region in the kernel address space of the host. In other words, to do a send, the host must move the data to the DMA region first, then notify the Lanai that the data is present, then the Lanai does the DMA from the DMA region to its SRAM.[Pakin95]

### *2.3.3 Messaging Layers*

The magnitude of the overhead involved in current messaging layer implementations can be seen in experiments conducted in this research's initial assessment of the Myrinet hardware and software. Using TCP/IP as the primary messaging layer, we found that the time to send a 0 byte message from a host to itself was approximately 369 microseconds. This test approximates the time for a message to traverse the messaging layers. During this time, 472 bytes could be sent across a Myrinet link operating at the peak hardware bandwidth of 1.28 Gbps. Obviously, the speed of the messaging layers has not reached the speed of the network hardware. Because of this performance gap, there are many groups writing "leaner" messaging implementations such as University of California's at Berkeley's Active Messages,[vonEiken92] University of Illinois' Fast Messages,[Pakin95] Mississippi State University's Bulldog Messages,[Henley97] and Tsukuba Research Center's PM.[Tezuka96]

One messaging layer implementation for Myrinet is Berkeley's Active Messages (AM) project.[vonEiken92] The idea behind AM is that computation and communication can be overlapped by adding functionality to the messaging layer. This overlapping can hide the overhead of the messaging layers. The mechanism for the overlapping is to include in the header of a message the address of an instruction sequence within the running parallel program which reads in the message data, allowing the program to continue execution. In TCP/IP, the operating system would have to switch to supervisor mode and get the data from the network interface buffer. AM takes the opposite approach and puts the data where it is needed by the program. This is a paradigm change in messaging from a pull of data by the higher layers to a push of data by the lower layers. There will be a public release of AM called AM-II by the end of 1997 and an MPI implementation on top of AM-II will be included. The Berkeley project uses Sparc Ultras with Solaris for their NOW, so new versions should be compatible with the AFIT NOW.

Piggybacking on the AM model, Illinois Fast Messages (FM) was developed.[Pakin95] FM uses the message handling functions of AM as described earlier, but extends the functionality of the messaging layer to increase performance. FM takes advantage of the high reliability of the network to make stronger guarantees on the hardware. These guarantees include reliable, ordered delivery and communication scheduling. The first guarantee is easy to realize with a reliable network. The second guarantee allows a sender to keep sending even when receivers have not extracted the messages from the NIC. This requires more buffering on the receive side and may slow down operation of the messaging layer. There is a tradeoff between more guarantees of

the layers versus higher throughput without buffering. Another interesting design tradeoff in FM is the communication mechanism between the Lanai and the host processor. FM uses programmed I/O for sending from the host to the Lanai, but DMA from the Lanai to the host memory space. There is an MPI implementation for the Sparc/Solaris platform, however new versions of FM and MPI-FM will be limited to Windows NT and Linux, PC-based platforms.

Another variation of the AM model is PM (abbreviation origin unknown) developed by Tsukuba Research Center in Japan.[Tezuka96] PM implements a different flow control mechanism than FM, but guarantees reliable, in-order delivery like FM. It's most important contribution seems to be network context switching which allows multiple processes to use the Myrinet interface. FM and AM do not yet provide this context switching and as a result, only one process can access the Myrinet network. Unfortunately, PM was developed for the SunOS operating system, and for Linux which are not compatible with the AFIT NOW.

Finally, a newer messaging layer is Bulldog Messages (BDM) from Mississippi State University.[Henley97] BDM is similar in many ways to the FM project, however BDM offers different levels of reliability in an attempt to achieve greater performance. BDM has better error-recovery for events such as lost packets due to a cable being unplugged. In the design of BDM, it was determined that programmed I/O only was the best choice for communication from the host processor to the Lanai. The designers believed the DMA setup time would overshadow the performance gains of the SBus burst DMA

transfer mode. Besides offering an MPI implementation on top of BDM, there is also a package called Bulldog Threads (BDT) which gives a programmer a thread interface with which to program. BDM and BDM-MPI are being developed for the Sparc/Solaris platform and in the future, the Windows NT/PC platform.

Table 2-1 shows the performance of TCP/IP, AM, FM, PM and BDM on Myrinet. These numbers should not be taken at face value because they were run on different workstations with different operating systems on different network benchmark programs. It does, however, provide a rough measure of both the throughput and latency that can be achieved on Myrinet using these particular messaging implementations.

*Table 2-1. Latency and Throughput of Messaging Layers for Myrinet.*

| Messaging Layer | Maximum Throughput | Minimum Latency |
|---|---|---|
| **TCP/IP** | 11 MB/sec | 369 microseconds |
| **AM** | 28 MB/sec | 16.1 microseconds |
| **FM** | 17 MB/sec | 22 microseconds |
| **PM** | 32 MB/sec | 24 microseconds |
| **BDM** | 42 MB/sec | 19 microseconds |

### *2.3.4 MPI Communication Libraries*

Another important aspect of communication performance in parallel computing is the communication library used by the programmer. In the past, each platform, message passing or shared memory, had its own proprietary communication library. As a result, programs were not portable from platform to platform. The advantage was that the communication library was fine-tuned for the particular platform. In an attempt to realize

portability, a standard communication library was developed called the Message Passing Interface (MPI).[MPI94] The goal of MPI is to be a standard and efficient message passing library with maximum functionality. MPI is a standard library of functions for which there are many implementations for various platforms. Some of these implementations include CHIMP,[CHIMP97] LAM,[LAM97] Unify,[Unify97] and MPICH.[MPICH97] MPICH is one of the most popular implementations because all of the MPI functions are broken down into combinations of 13 lower level functions called the Abstract Device Interface (ADI).[Lauria96] This simplicity makes implementation straight forward because only these 13 functions need to be created for a given platform.

### *2.4 MPPs Used in Research*

To compare the performance of NOWs to MPPs, experiments were performed on two available MPPs, the IBM SP2 and the Intel Paragon. The specifics of these two platforms are given here.

#### *2.4.1 IBM SP2*

The IBM SP2 used in this research is located at the Aeronautical System Center's Major Shared Resource Center (MSRC) at Wright-Patterson AFB.[MSRC97] The MSRC SP2 has 256 nodes comprised of the RS/6000 Power2 SC (P2SC) processor operating at 135 MHz. There are 233 compute nodes available, each with 1 GB of main memory. The P2SC is a 4-issue superscalar processor which can perform two simultaneous integer and FP instructions. Also, the P2SC can perform an unusual FP multiply/add instruction

in one cycle. There is a 128 KB data cache which is 4-way set associative and a 32 KB instruction cache. The NIC card on each node has a Power PC 601 processor and performs DMA only to and from the host processor. The DMA performance varies, with a maximum transfer rate of 160 MB/sec on the 64 bit, 20 MHz microchannel bus.[IBM97]

The interconnection network for the SP2 is a multistage Omega network. Each group of 8 nodes are connected via a switch board called a frame which is comprised of 4 4x4 Omega switches. Multiple frames are interconnected to scale the network with intermediate switching hardware. The network has a peak, theoretical bandwidth of 300 MB/s in full-duplex mode.[IBM97]

### *2.4.2 Intel Paragon*

The Intel Paragon used in this research is located at Rome Laboratory's (RL) High Performance Computing Facility. The RL Paragon has 321 nodes, of which 272 are available for computation. Each compute node has three i860 XP processors, one for communication and two for computation. The i860 is a older, non-superscalar, RISC processor. There are 32 KB on-chip data and instruction caches and there is 64 MB of main memory which is shared between the processors.[Intel95]

The nodes are interconnected in a 16x17 node mesh without wraparound connections. The nodes connect to a Message Router Chip with has five ports, one for each direction (except the edges) and one for the node. The computation and communication processors communicate through the shared memory which forces a

cache consistency policy.[Hennessy96] There is a maximum theoretical bandwidth of 400 MB/sec in full duplex mode on the Paragon backplane.[Intel95]

## *2.5 Related Research*

There has been research in both NOW and MPP performance and 2-D FFT algorithms. This section describes some of this research and discusses the differences between the results presented in these papers and this research effort.

### *2.5.1 NOWs*

In the area of NOW performance, there are various papers comparing NOWs to other platforms for a variety of applications. An example of recent research which uses newer networks is found in [Judd97]. This paper presents results using Ultra Sparc's connected via 100 Mbps Fast Ethernet and 155 Mbps ATM and an older IBM SP2 with 62.5 MHz processors for a signal processing application (not an FFT) called P-CLUSTER. This research showed their NOW outperformed the SP2, however they used only TCP/IP as the messaging layer which clouds the results. Using a more efficient messaging layer and a newer SP2, the results would be expected to favor the SP2. One interesting conclusion from this paper was that Fast Ethernet outperformed ATM because of its lower messaging overhead.

A more general comparison of parallel platforms for a variety of applications can be found in the Numerical Aerodynamic Simulation (NAS) benchmarks.[NAS94] The NAS

benchmarks are a set of eight problems that were developed to study the performance of parallel computers, especially in the area of computational fluid dynamics. The solutions to these eight problems are implemented by various supercomputer vendors using High Performance Fortran or C/MPI. One of the applications is a 3-D FFT solution to a partial differential equation. For an SP2 and a Paragon configured similarly to the ones used in this research, the SP2 was 75% faster than the Paragon for the only common input size and number of processors. In this application, hand-tuned assembly code may be used for the FFT kernel which is an exception to the normal NAS framework and is different than the framework of this research. Also, the NAS benchmarks report only runtime results, so the speedup metric (versus serial implementations) is not provided as it is in this thesis effort. Currently, the NAS results do not provide a benchmark of a NOW using Myrinet or the available efficient messaging layers. NAS results are updated quite often and are available on-line.[NASWeb97]

### *2.5.2 2-D FFT Research*

The 2-D FFT has been a well-studied topic both for algorithmic efficiency and parallel implementation. A paper by Patel and Jamieson compares the row-column and vector-radix algorithms on the MasPar MP-1 (mesh), Intel iPSC 860 (hypercube) and the Intel Touchstone Delta (precursor to the Paragon). [Patel93] On the Delta, the row-column is shown to be the better algorithm for larger per node problem size (N/p) while the vector-radix is superior at small N/p. Also, the results support the isoefficiency analysis which states that the 2-D FFT is not scaleable on a mesh architecture. The paper does not

detail the programming language, communication library or any part of the comparative framework, so the timing results cannot be directly compared to this research.

Another paper, by An et al., compares the row-column and vector-radix algorithms on the Intel Paragon. [An95] The row-column and two version of the vector-radix using two different data distribution schemes are described. The partial vector-radix which is the version of the vector-radix used in this research is shown to be superior to the full, checkerboard vector-radix and also the row-column. The 2-D FFT implementations were written in Fortran and used hand-tuned assembly coded routines for FFT computations and matrix transpositions. This framework is different than the one used in this research which is limited to portable, C and MPI-based, implementations.

# *III. Methodology*

Chapter II gave the background necessary to understand the differences between MPPs and NOWs. In order to quantify those differences, experiments were performed to measure the speed and efficiency of the different platforms. The application chosen to be the comparison tool was the two-dimensional fast Fourier transform (2-D FFT). The 2-D FFT was chosen for two reasons. First, this research is under the **C**ommon **H**igh-Performance Computing **S**oftware **S**upport **I**nitiative (CHSSI).[CHSSI97] The CHSSI program was designed to provide the Department of Defense with high performance, scaleable software applications for today's computational platforms. The 2-D FFT falls under the Signal/Image Processing Computational Technology Area defined in the CHSSI program. Second, the 2-D FFT was chosen because it represents a class of applications in which there is a regular matrix data structure, a regular decomposition of the data, and a regular communication pattern.

The purpose of this chapter is two-fold. First, the 2-D FFT is detailed. The theoretical background of the 1-D and 2-D FFTs and their derivations are shown. Next, the 2-D FFT algorithms are discussed, and detailed descriptions of the implementations of these algorithms are given. Also, the theoretical complexity of the 2-D FFT is described. The second part of this chapter outlines the experimental methodology used to compare the AFIT NOW with the SP2 and Paragon. The degrees of freedom of the experiments as well as the platform variations which may effect performance are discussed.

### *3.1 1-D FFT*

To understand the FFT, the building blocks from which it is derived must be understood. Essentially, the Fourier transform is a mapping from the time-domain to the frequency domain. It defines a continuous signal in the time domain by an infinite series of sine and cosine functions. The frequency of the sinusoids form the frequency content of the time-domain signal. Quite often, the frequency content is more valuable or has mathematical properties which allow simpler computation than the time-domain representation. An obvious example is filtering which requires only simple arithmetic in the frequency domain, but much more complicated convolution in the time-domain.

#### *3.1.1 1-D FFT Theory*

Since continuous and infinite signals cannot be represented using a digital computer, the time-domain signal is sampled at a regular interval or frequency. As a result, the pure Fourier transform, which is an integral, is changed to the discrete Fourier transform (DFT), which is a summation of the sampled signal. The DFT is expressed mathematically as [ODonnell96]

$$X(k) = \sum_{n=0}^{N-1} x(n) W_N^{nk}, \qquad \text{k=0,1,...N-1} \qquad (1)$$

where $W_N = e^{-j2\pi/N}$.

A direct implementation of the 1-D DFT results in a $O(N^2)$ algorithm because each of the N points is the sum of all N other points. A more efficient implementation of the DFT called the Fast Fourier Transform (FFT) was developed by Cooley and Tukey in 1965.[Cooley65] They observed that the summation in the DFT could be broken into two summations of N/2 points. This decimation continues until only two complex points remain in the computation, which is called a twiddle. Because the decimation halves the input and finally results in a two point twiddle, this is called a Radix-2 FFT. The result of this decomposition is an O(NlogN) FFT which significantly reduces the number of computations compared to the DFT, especially when N is large.

The decimation of the DFT can be performed in two equivalent ways. Decimation-in-time (DIT) divides the *input* sequence into even and odd components (assuming radix-2). The resulting flow graph of a DIT 8 point FFT is shown in Figure 3-1.

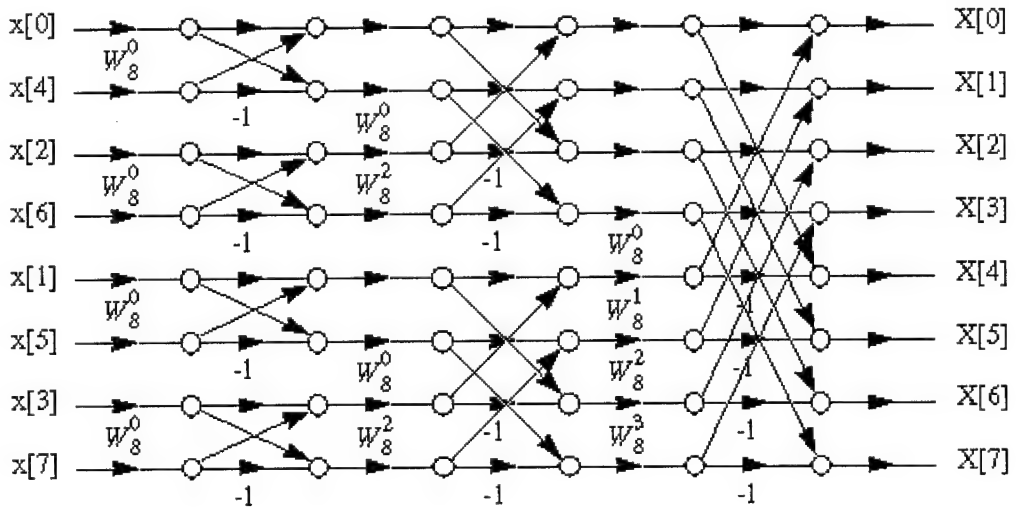


*Figure 3-1. 8 Point Radix-2 DIT FFT. [ODonnell96]*

The decimation-in-frequency (DIF) FFT divides the *output* sequence into even and odd components. Figure 3-2 shows an 8 point radix-2 DIF FFT.

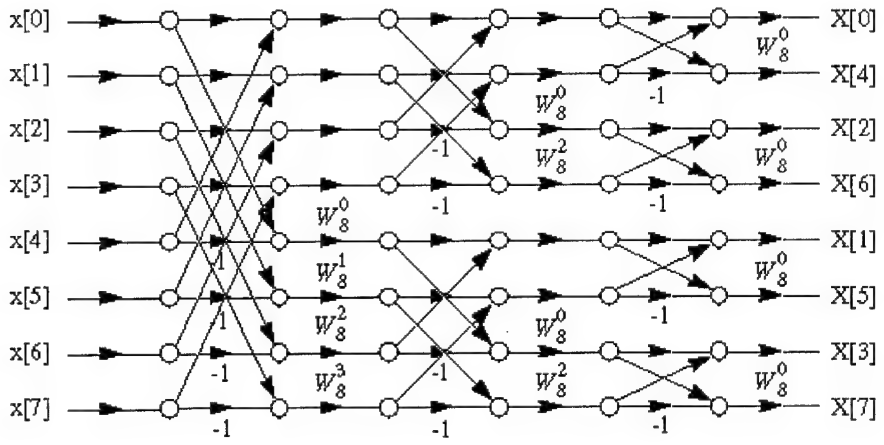


*Figure 3-2. 8 Point Radix-2 DIF FFT. [ODonnell96]*

Much research has been done to improve the basic radix-2 FFT developed by Cooley and Tukey. One improvement was to increase the radix to higher powers of two. The result was radix-4, 8, 16, etc. algorithms. These algorithms reduce the algorithmic complexity from $O(N\log_2 N)$ to $O(N\log_R N)$ where R is the radix. The higher radix algorithms require less stages, less multiplications and less additions, but have a more complex butterfly structure. There are also diminishing returns to using higher radices because the number of additions and multiplications decrease at a slower rate. Another disadvantage of higher radix algorithms is that the number of possible input sizes is reduced because the input must be a power of R instead of only a power of two.

A relatively new algorithm, called split-radix has been developed which incorporates the advantages of radix-2 and higher radix algorithms.[Duhamel84] The basic

split-radix algorithm divides the input into odd and even indices just as radix-2 algorithms. The even indices are treated the same as in radix-2, however the odd indices are evaluated using the radix-4 algorithm. The advantage of evaluating the odd terms using radix-4 decomposition is that the number of complex multiplications and additions are reduced. It has been found that similar to higher-radix algorithms, there are diminishing returns with splits other than radix-2/4. Also, all power-of-two input sizes are easily supported with split-radix, just as with radix-2. Figure 3-3 shows the number of operations required for radix-2, radix-4 and split radix 1-D FFT algorithms for some of the input sizes studied in this research. Note that radix-4 is not shown for non-powers of 4 (128 and 512).

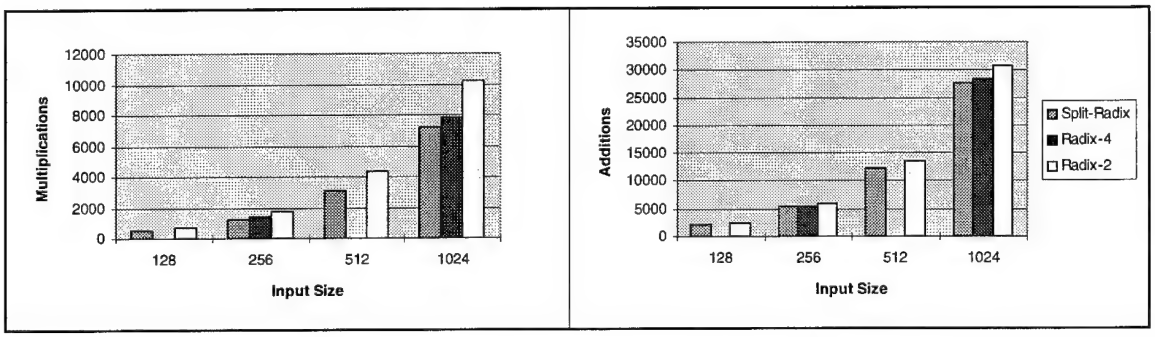


*Figure 3-3. Number of Multiplication and Additions for 1-D FFT Algorithms.*

There are other algorithms which have been developed which improve upon the Cooley-Tukey algorithm, but are not discussed in depth here due to the time constraints of this research. These algorithms include, the prime-factor algorithm,[Kolba77] the Hartley transform,[Bracewell86] and the Winograd transform.[Heideman86]

### *3.1.2 1-D FFT Implementations*

The goal of this research was not to develop 1-D FFT code, but rather to use existing 1-D FFT code whenever possible. Various 1-D routines were found on the Internet and tested for accuracy as well as performance. Detailed results of the 1-D FFT analysis can be found in [Lamont97], but for brevity, the results are summarized here. There were two clear winners among the 1-D codes tested: FFTW and the Japanese Split-Radix.

#### *3.1.2.1 FFTW*

The Fastest FFT in the West (FFTW) is a completely different approach to programming than the other FFTs studied in this research.[Frigo97] FFTW writes code at run-time based on parameters it determines through experimentation. The theory behind FFTW is that efficient use of the memory hierarchy will determine the performance of an FFT. To maximize cache performance, FFTW uses recursion until the "bottom-out" point fits in the cache. Since cache effects are rarely predictable, FFTW will test different "bottom-out" points to find the optimal decomposition by generating code for each case. For example, an 8 point 1-D FFT could be broken down into 4 2-point FFTs, 2 4-point FFTs, or 1 8-point FFT. Once an optimal decomposition is found by executing and timing the different FFTs, a "plan" is created which can then be used for FFTs of that size from that point in the program. The code generated uses known compiler optimizations for each supported platform to write the most efficient code possible.

The FFTW approach has advantages for research use, but its disadvantages outweigh them for production use. The advantage of the FFTW approach is that it has consistently good performance in terms of runtime across all platforms and various cache sizes. In some ways, FFTW has portability because of its competitive performance and flexible code generation. However, if a new platform, compiler or compiler optimization is introduced, FFTW must be modified (only by its writers, given its complexity). A disadvantage of FFTW is its time overhead in testing various sizes (especially large sizes) and writing the code. If "standard" code is competitive with FFTW's performance, the standard code would be more attractive for production use.

#### *3.1.2.2 Japanese Split-Radix FFT*

A split-radix FFT obtained from the University of Tokyo, Japan, performed the best of the "standard" (code pre-determined) implementations that were tested in terms of runtime.[Ooura97] There are three reasons for the Japanese code's performance. The first reason is compiler optimized code. The Japanese code performs loop invariant code removal, common sub-expression elimination and other techniques which help compilers to optimize code. The second reason for the code's performance is its suitability to the memory hierarchy of current processors. The code is written to maximize data locality and hence increase cache hits. The third reason for the code's performance is its use of the computationally efficient split-radix algorithm. The split-radix algorithm minimizes the number of multiplications and additions over radix-R FFTs as discussed in Section 3.1.

### *3.2 2-D FFT*

The 2-D FFT is an extension of the 1-D FFT into a second dimension and is commonly used for signal and image processing. The 2-D FFT is derived from the 2-D DFT which is described mathematically as [ODonnell196]

$$X(k_1,k_2)=\sum_{n=0}^{N_1-1}\sum_{n_2=0}^{N_2-1}x(n_1,n_2)W_{N_1}^{n_1k_1}W_{N_2}^{n_2k_2}\ , \qquad (2)$$

where $W_{N_j}=e^{-i/2\pi/N_j}$ , j=1,2,... .

The time-domain input signal $x(n_1,n_2)$ is transformed to a frequency domain signal $X(k_1,k_2)$. The straightforward 2-D DFT has complexity $O(N^4)$, but it can be reduced to $O(N^2\log N)$, in a similar manner as the 1-D DFT is reduced to a 1-D FFT. There are two primary 2-D FFT algorithms which arise from two different data decompositions, the row-column method and the vector-radix method.

#### *3.2.1 2-D FFT Algorithms*

This section describes two fundamental 2-D FFT algorithms. These descriptions are implementation-independent and give an overview of the algorithm.

##### *3.2.1.1 Row-Column Algorithm*

The first algorithm is the row-column method. Based on the mathematical properties of the 2-D FFT, the computation can be decomposed into two steps. In the

first step, 1-D FFTs are performed on the rows of the image. In the second step, 1-D FFTs are performed on the columns of the image. This decomposition can be expressed mathematically as [Pitas93]

$$X'(n_1,k_2) = \sum_{n2=0}^{N2-1} x(n_1,n_2) W_{N2}^{n2k2}$$
$$X'(k_1,k_2) = \sum_{n2=0}^{N2-1} X'(n_1,k_2) W_{N1}^{n1k1} \qquad (3)$$

### *3.2.1.2 Vector-Radix Algorithm*

The second 2-D FFT algorithm is the vector-radix method. This algorithm is an extension of the Cooley-Tukey 1-D FFT into a second dimension. Just as the Cooley-Tukey method successively breaks down a 1-D FFT into smaller sizes until two elements remain, the basic vector-radix method breaks down the 2-D FFT into smaller sizes until four elements remain, two in each dimension. Mathematically, the vector-radix algorithm is expressed as [Pitas93]

$$X(k_1,k_2) = \sum_{n1=0}^{N-1}\sum_{n2=0}^{N-1} x(n_1,n_2) W_N^{n1k1} W_N^{n2k2}\ , \qquad (4)$$

which can be divided into four summations,

$$X(k_1,k_2) = G_{ee}(k_1,k_2) + G_{eo}(k_1,k_2) + G_{oe}(k_1,k_2) + G_{OO}(k_1,k_2)\ , \qquad (5)$$

where

$$G_{ee}(k_1,k_2) = \sum_{l_1=0}^{N/2-1}\sum_{l_2=0}^{N/2-1} x(2l_1,2l_2)\, W_N^{2l_1k_1} W_N^{2l_2k_2}$$

$$G_{eo}(k_1,k_2) = \sum_{l_1=0}^{N/2-1}\sum_{l_2=0}^{N/2-1} x(2l_1,2l_2+1)\, W_N^{2l_1k_1} W_N^{2l_2k_2}$$

$$G_{oe}(k_1,k_2) = \sum_{l_1=0}^{N/2-1}\sum_{l_2=0}^{N/2-1} x(2l_1+1,2l_2)\, W_N^{2l_1k_1} W_N^{2l_2k_2} \qquad (6)$$

$$G_{oo}(k_1,k_2) = \sum_{l_1=0}^{N/2-1}\sum_{l_2=0}^{N/2-1} x(2l_1+1,2l_2+1)\, W_N^{2l_1k_1} W_N^{2l_2k_2}$$

Figure 3-4 shows the four element butterfly structure that results from the vector-radix decomposition.

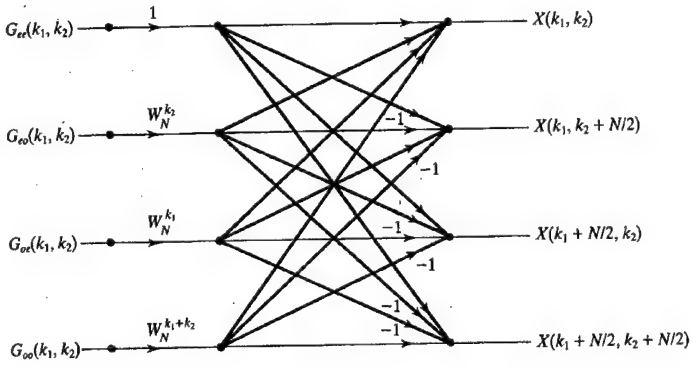


*Figure 3-4. Vector-Radix Butterfly. [Pitas93]*

The access pattern of a 2-D matrix during the vector-radix algorithm clearly shows the regular data decomposition. The twiddle consists of points which are 1, 2, 4, ...N/2 points apart in each of the $\log_2 N$ stages. Figure 3-5 shows a single twiddle for each stage of the decimation in time algorithm. The shaded elements show the coordinates accessed.

Stage 0

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| (0,0) | (0,1) | (0,2) | (0,3) | (0,4) | (0,5) | (0,6) | (0,7) |
| (1,0) | (1,1) | (1,2) | (1,3) | (1,4) | (1,5) | (1,6) | (1,7) |
| (2,0) | (2,1) | (2,2) | (2,3) | (2,4) | (2,5) | (2,6) | (2,7) |
| (3,0) | (3,1) | (3,2) | (3,3) | (3,4) | (3,5) | (3,6) | (3,7) |
| (4,0) | (4,1) | (4,2) | (4,3) | (4,4) | (4,5) | (4,6) | (4,7) |
| (5,0) | (5,1) | (5,2) | (5,3) | (5,4) | (5,5) | (5,6) | (5,7) |
| (6,0) | (6,1) | (6,2) | (6,3) | (6,4) | (6,5) | (6,6) | (6,7) |
| (7,0) | (7,1) | (7,2) | (7,3) | (7,4) | (7,5) | (7,6) | (7,7) |

Stage 1

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| (0,0) | (0,1) | (0,2) | (0,3) | (0,4) | (0,5) | (0,6) | (0,7) |
| (1,0) | (1,1) | (1,2) | (1,3) | (1,4) | (1,5) | (1,6) | (1,7) |
| (2,0) | (2,1) | (2,2) | (2,3) | (2,4) | (2,5) | (2,6) | (2,7) |
| (3,0) | (3,1) | (3,2) | (3,3) | (3,4) | (3,5) | (3,6) | (3,7) |
| (4,0) | (4,1) | (4,2) | (4,3) | (4,4) | (4,5) | (4,6) | (4,7) |
| (5,0) | (5,1) | (5,2) | (5,3) | (5,4) | (5,5) | (5,6) | (5,7) |
| (6,0) | (6,1) | (6,2) | (6,3) | (6,4) | (6,5) | (6,6) | (6,7) |
| (7,0) | (7,1) | (7,2) | (7,3) | (7,4) | (7,5) | (7,6) | (7,7) |

Stage 2

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| (0,0) | (0,1) | (0,2) | (0,3) | (0,4) | (0,5) | (0,6) | (0,7) |
| (1,0) | (1,1) | (1,2) | (1,3) | (1,4) | (1,5) | (1,6) | (1,7) |
| (2,0) | (2,1) | (2,2) | (2,3) | (2,4) | (2,5) | (2,6) | (2,7) |
| (3,0) | (3,1) | (3,2) | (3,3) | (3,4) | (3,5) | (3,6) | (3,7) |
| (4,0) | (4,1) | (4,2) | (4,3) | (4,4) | (4,5) | (4,6) | (4,7) |
| (5,0) | (5,1) | (5,2) | (5,3) | (5,4) | (5,5) | (5,6) | (5,7) |
| (6,0) | (6,1) | (6,2) | (6,3) | (6,4) | (6,5) | (6,6) | (6,7) |
| (7,0) | (7,1) | (7,2) | (7,3) | (7,4) | (7,5) | (7,6) | (7,7) |

*Figure 3-5. Matrix Access of 8x8 Vector-Radix 2-D FFT.*

As in the 1-D case, the vector-radix can use decimation in time or decimation in frequency. Also, the radix can be extended to larger sizes (i.e. radix-2, 4, 8, etc.) and a split radix decomposition can be performed to reduce the number of complex multiplications.[Chan92]

### *3.2.2 2-D FFT Implementations*

This section describes three implementations of the two 2-D algorithms outlined in the previous section. The row-column and pipeline are two different implementations

of the row-column algorithm and the vector-radix is the only implementation of the vector-radix algorithm.

### *3.2.2.1 Row-Column Implementation*

This implementation of the row-column algorithm is a straightforward mapping of parallel communication constructs onto the algorithm. The original source code was obtained from the Maui High Performance Computing Center and was subsequently modified.[Gusciora96] The NxN complex point image is initially read in by a single source processor. The image is then scattered to all the processors (including the source processor) by sending an (N/p)xN slice to the p processors. Figure 3-6 shows the source processor scattering the rows of the input image to the processors. All the figures shown for this implementation assume an 8x8 image size with four processors. The coordinates of each complex point are shown to better illustrate the movement of data at each stage.

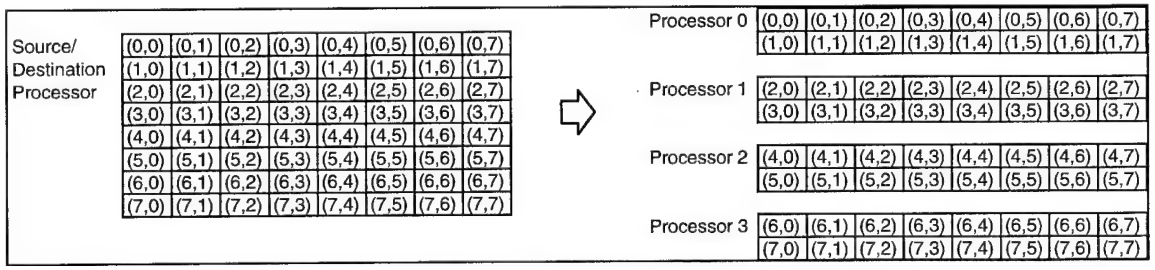


*Figure 3-6. Scatter of 8x8 Image in Row-Column 2-D FFT.*

Next, each processor performs (N/p) 1-D FFTs, one on each row of their slice. After the FFTs, the image must be transposed among the processors. This transposition is achieved by dividing each slice into p "chunks" of size (N/p)x(N/p), each of which are

sent to a different processor. This step uses the all-to-all communication pattern in which each processor sends to and receives messages from all other processors. The individual chunks must also be transposed themselves in order to insure all processors have data containing columns of the original image. These columns are stored as rows to facilitate sequential memory access. Figure 3-7 shows a single processor dividing it's slice into chunks (Step 1) and then transposing the individual chunks (Step 2) before performing the all-to-all communication (Step 3). All processors perform the first two steps, but only one is shown for clarity.

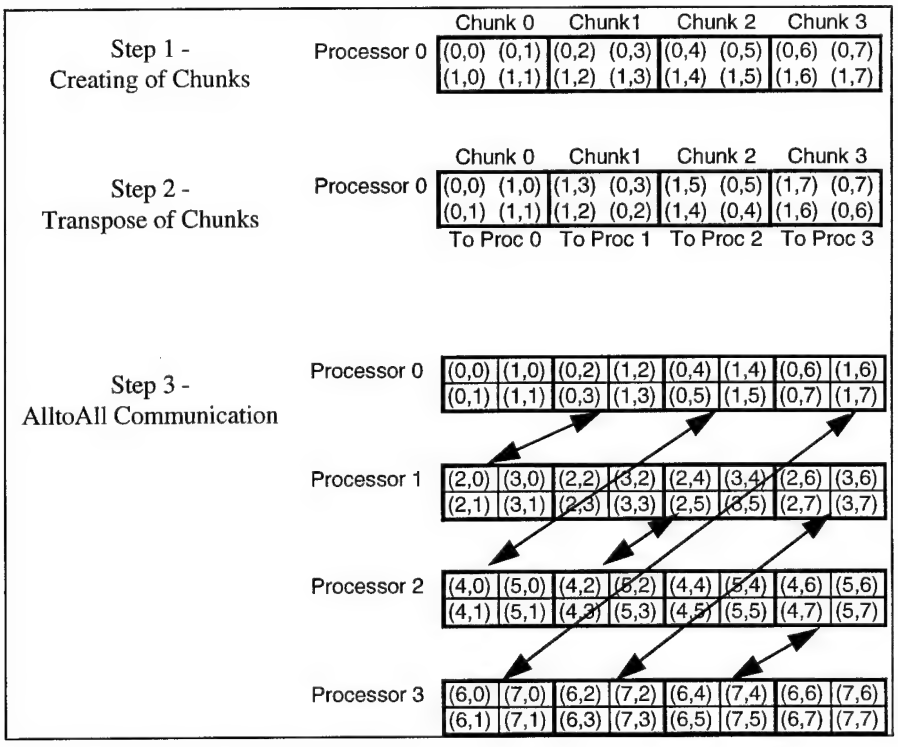


*Figure 3-7. Matrix Transposition Steps of Row-Column 2-D FFT.*

Next, the processors perform (N/p) 1-D FFTs, one on each column of the image. After the FFTs, the destination processor gathers the p slices back into a NxN image. The

final, transformed image must be completely transposed in order to put it back into the original order. Figure 3-8 shows the destination processor gathering the columns of the transformed image. Note, the figure does not show the final matrix transposition step.

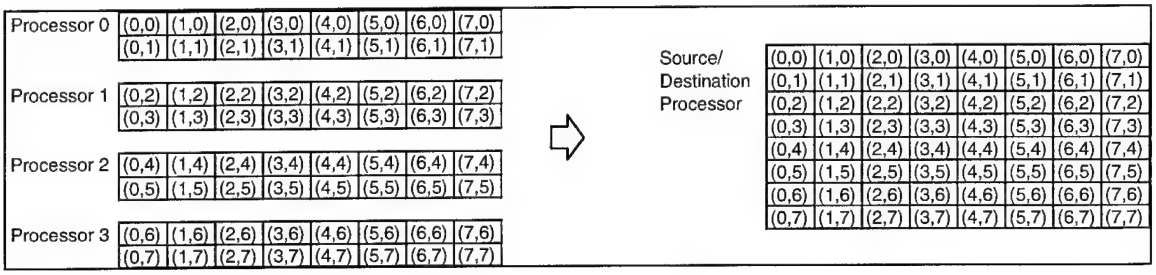


*Figure 3-8. Gather Step of Row-Column 2-D FFT.*

### *3.2.2.2 Vector-Radix Implementation*

The implementation of the vector-radix 2-D FFT is more complicated than the row-column implementation because of the more complicated algorithm. First, the image is read in by the source/destination processor. Next, a bit reversal of the image along the rows and the columns must be performed. This means that each point (i,j) must be moved to position (br(i),br(j)) where br(x) returns the bit-reversed value of x. Figure 3-9 shows the input matrix and the bit-reversed matrix for an 8x8 point image.

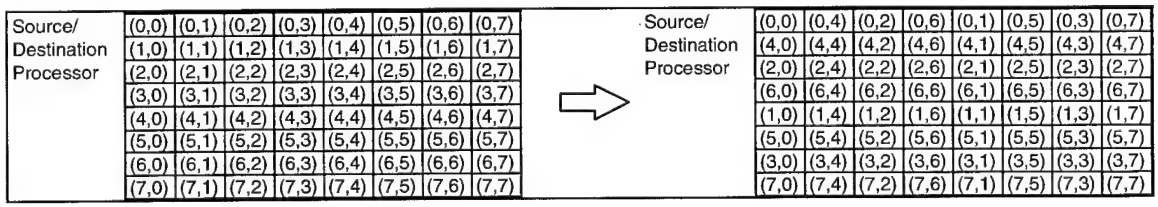


*Figure 3-9. Bit Reverse Step of Vector-Radix 2-D FFT.*

After the bit-reverse, the NxN image is scattered to all p processors just as in the row-column implementation. Next, the processors perform ($\log_2 n - \log_2 p$) stages of the FFT without any communication. During these stages, each processor is accessing elements that are less than (N/p) elements apart, so all the elements in the twiddle are stored locally. The final ($\log_2 p$) stage require elements in the twiddle which are not stored locally at each processor, so communication is necessary.

At each of the $\log_2 p$ communication stages, each processor swaps half of its slice with another processor. The destination processor is determined by adding $2^{\text{communication stage}}$ to the processor number and reducing modulo p, where the communication stage is 0, 1, ... $\log_2 p - 1$. Figure 3-10 shows the communication pattern for 8 processors. For example, processor 0 swaps with processor 1, then 2, then 4 during the three communication stages.

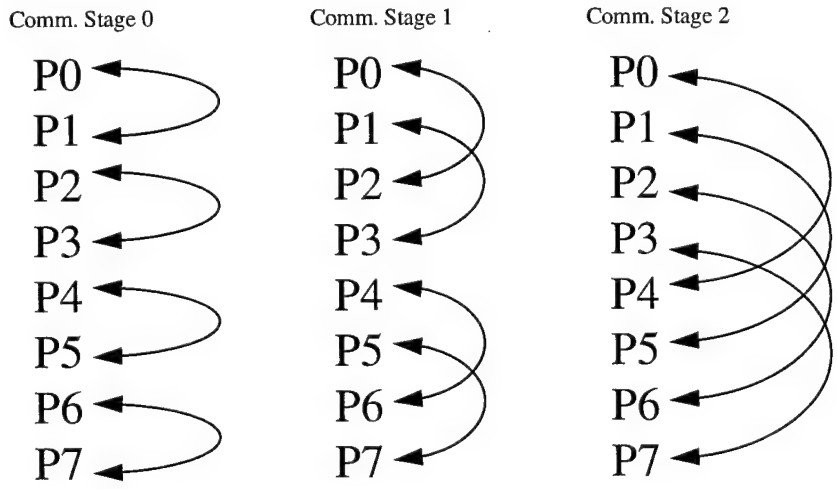


*Figure 3-10. Communication During Vector-Radix 2-D FFT with 8 Processors.*

During a communication stage, once the swapping takes place, the processor can complete the FFT of that stage by evaluating the twiddles. Figure 3-11 shows the matrix representation of this situation with similarly shaded blocks in the same twiddle. The points in the twiddles before communication are beyond processor slice boundaries. The figure also shows processor 0's slice after it has swapped it's lower half with processor 1's upper half. Note that after the swap, the twiddles access points two elements apart in the rows and four elements apart in the columns.

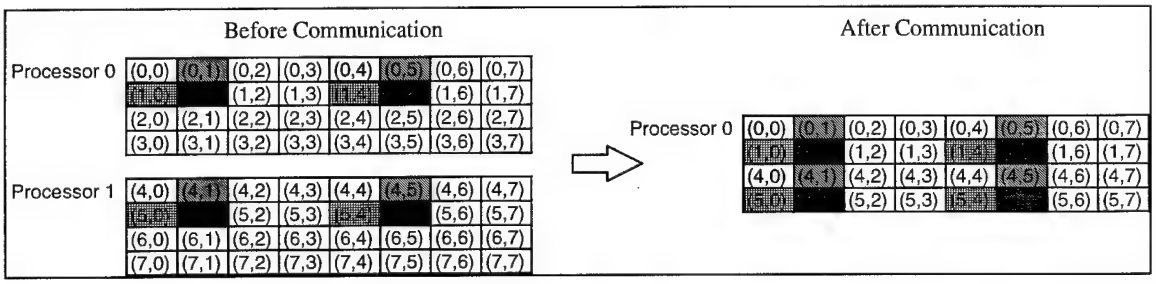


*Figure 3-11. Matrix Access Pattern During Communication Stage of Vector-Radix 2-D FFT.*

Finally, when all the FFT stages are complete, the image slices are gathered by the destination processor. The final image must be permuted, however, because of the swapping of rows which takes place during the communication stages. The upper half of each received slice moves to the (p*slice/2) row and the lower half of each slice moves to the (p*slice/2+N/2) row to return the image to its original order. Figure 3-12 shows the gather and permutation operations in one step.

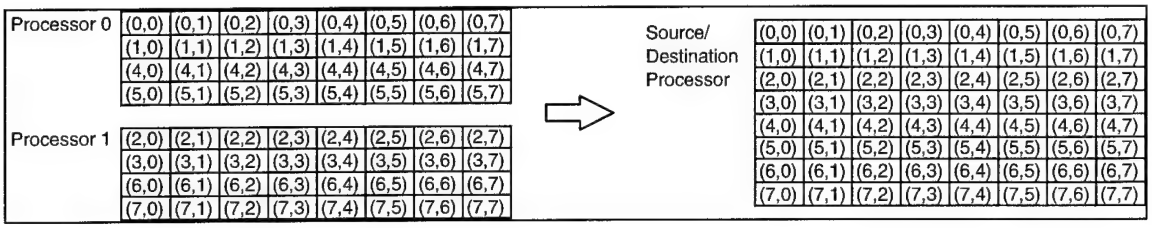


*Figure 3-12. Gather and Permutation Step of Vector-Radix 2-D FFT.*

### *3.2.2.3 Pipeline Implementation*

A third implementation of the 2-D FFT, called the pipeline, has been developed which uses the row-column method with a different division of labor among the processors. This approach was originally extracted from source code from the University of Southern California, but the algorithm as presented here is a modification of the original. [Su96] The pipeline method assumes that there would be a real-time source generating the original, untransformed image. This assumption allows the source processor to read the image from memory quickly as opposed to accessing a local or network disk drive. The algorithm uses three sets of processors, a single source/destination processor, a group of row processors and a group of column processors.

The only duties of the source/destination processor are to send out images (fill the pipe) and receive images (drain the pipe). The basic, steady-state algorithm works as follows.

**Source/Destination Processor**: The source/destination processor will receive $(N/p_2)$ columns from the $p_2$ column processors and then send $(N/p_1)$ rows of the NxN size image to the $p_1$ row processors.

**Row Processors**: The $p_1$ row processors receive $(N/p_1)$ rows from the source, perform $p_1$ 1-D FFTs on the $p_1$ rows and then send the rows to the column processors. The image must be transposed so that the column processors receive the columns of the original image. This transposition is achieved by the row processors dividing the $(N/p_1)$xN slice of the image into $p_2$ $(N/p_1)$x$(N/p_2)$ chunks. The column processors receive chunks from the row processors and assemble them into columns, which are stored in memory as rows.

**Column Processors**: The $p_2$ column processors then perform $p_2$ 1-D FFTs on the $p_2$ columns of the image. After the FFT, the $p_2$ column processors send their $p_2$ columns to the source/destination processors to complete a single iteration.

Because of the nature of the pipeline control structure, the pipe must initially be filled by the source/destination processor with two sends to the row processors. At the end, the source/destination processor drains the pipe by performing two receives from the column processors. Figure 3-13 shows four iterations of the pipeline algorithm and also illustrates the overlapping of communication and computation between the different sets of processors.

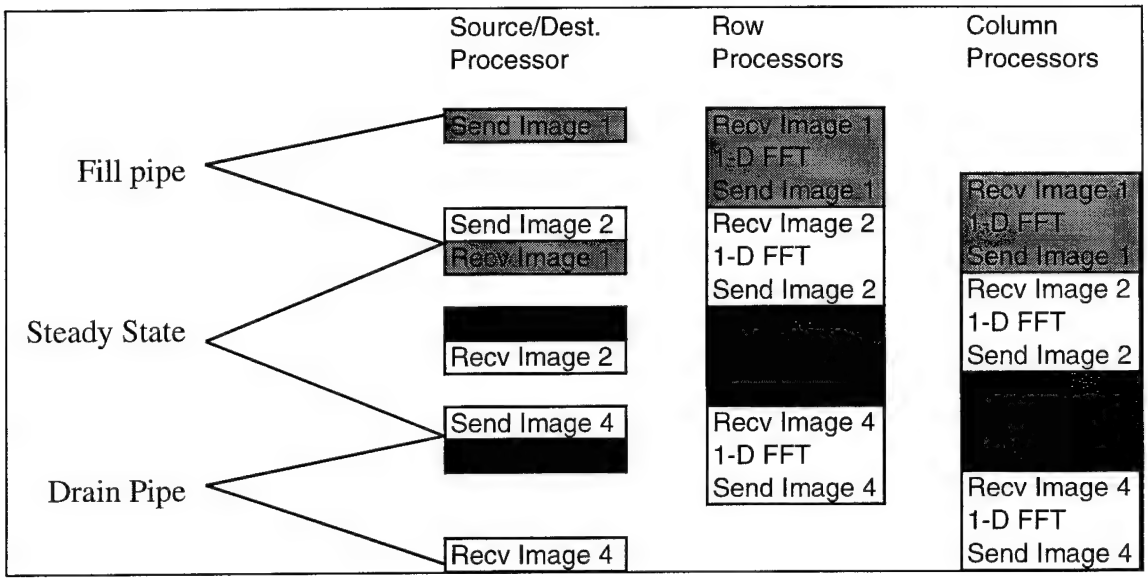


*Figure 3-13. Overview of Pipeline 2-D FFT.*

The movement of data in the pipeline is shown in Figures 3-14, 3-15 and 3-16. There are two row and two column processors. The coordinates of the matrices are shown to show the movement of individual points. Figure 3-14 shows the source/destination processor sending $n/p_1$ rows to the row processors. Figure 3-15 shows the row processors sending their chunks to the column processors. Finally, Figure 3-16 shows the column processors sending their $n/p_2$ columns to the source/destination processor.

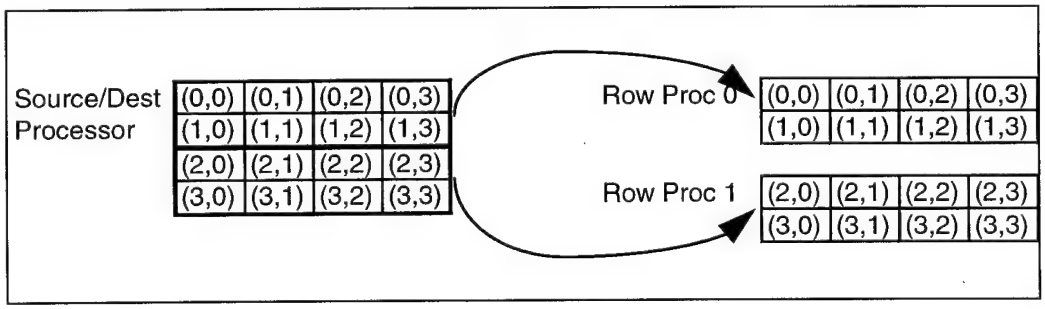


*Figure 3-14. Source/Destination Processor Sending to Row Processors.*

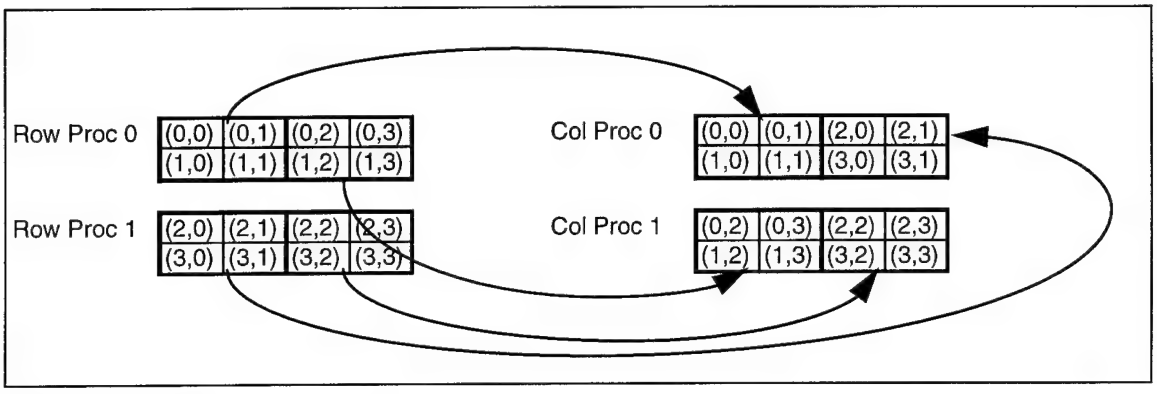


*Figure 3-15. Row Processors Sending to Column Processors.*

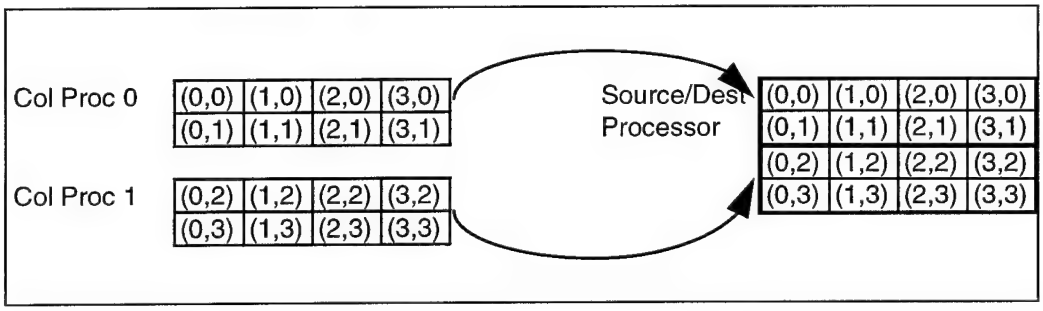


*Figure 3-16. Column Processors Send to Source/Destination Processor.*

### *3.2.3 Theoretical Complexity Analysis*

This section analyzes the theoretical complexity of the 2-D FFT. First, an isoefficiency analysis is given to determine the scalability of the 2-D FFT. Next, the communication and computational complexity of the three algorithms is compared. Finally, the pipeline is analyzed in more depth because its unique control structure makes theoretical analysis difficult.

#### *3.2.3.1 Isoefficiency Analysis*

The theoretical parallel runtime, as describe by Kumar et al., incorporates message startup time ($t_s$), per word transfer time ($t_w$), per hop transfer time ($t_h$) and computation

time ($t_c$) in order to approximate the asymptotic runtime.[Kumar94] This analysis is effective for comparing algorithms in the large input size, asymptotic case to approximate the scalability as more processors are added. The metric used to measure scalability is isoefficiency, which is the rate at which the problem size must be increased in order to keep constant efficiency. The isoefficiency varies depending on the algorithm and the communication mechanisms of the platform. For example, a broadcast operation on a mesh takes longer than on a hypercube because the number of links is less on the mesh.

The isoefficiency analysis of the algorithms will not be presented here for brevity. Interested readers should refer to [Kumar94] and [Lamont97]. The primary conclusion of the isoefficiency analysis is that the 2-D FFT is not very scaleable. For example, for the row-column algorithm, the isoefficiency on a hypercube (which could be approximated by a switching network such as Myrinet on the AFIT NOW or the SP2) is approximately $\frac{t_h}{t_c} p^2 \log p$, where p is the number of processors. The isoefficiency on a mesh (Paragon) would be $\frac{t_w}{t_c} p^{3/2}$ and on a bus (ethernet), $p^2$. These results show that the input size must be increased in a non-linear fashion to keep efficiency constant. Also, as the number of processors increases, the efficiency decreases dramatically. As a result, the 2-D FFT is not considered theoretically scaleable to large numbers of processors.

### *3.2.3.2 Computational Complexity*

In many ways, this asymptotic analysis falls short of describing the differences between the algorithm implementations at the given sizes in this research. For example, the asymptotic (order-of) analysis for the computation in the FFT is deceiving because the split-radix and radix-R algorithms are both O(NlogN). However, as discussed earlier, the split-radix is significantly more efficient in terms of the number of multiplications, for large input sizes. The use of non-blocking communication also blurs the time differences between computation and communication since they can be overlapped. An alternative to asymptotic parallel runtime is to analyze the message traffic and the actual number of computations performed.

The computational complexity can be estimated by the number of additions and multiplications necessary for the given algorithm. The row-column and pipeline have identical operations because they use the same split-radix 1-D FFT. Table 3-1 shows that the vector-radix requires fewer multiplications and additions, when compared to the row-column.

*Table 3-1. Number of Operations for Row-Column and Vector-Radix Implementations.*

| | Multiplications | | | Additions | | |
|---|---|---|---|---|---|---|
| N | Row-Column | Vector-Radix | % less | Row-Column | Vector-Radix | % less |
| 128 | 145408 | 86016 | 40.85% | 588800 | 229376 | 61.04% |
| 256 | 716800 | 393216 | 45.14% | 2750464 | 1048576 | 61.88% |
| 512 | 3399680 | 1769472 | 47.95% | 12578816 | 4718592 | 62.49% |
| 1024 | 15712256 | 7864320 | 49.95% | 56614912 | 20971520 | 62.96% |
| 2048 | 71270400 | 34603008 | 51.45% | 251641856 | 92274688 | 63.33% |

#### *3.2.3.3 Communication Complexity*

The communication of the algorithms can be measured by tallying the number of messages sent and the size of the messages. Figure 3-17 shows the number of messages sent by the three implementations on a logarithmic scale. The row-column sends the most messages as the number of processors increases. The pipeline sends the least messages until 32 processors, where the vector-radix sends the least.

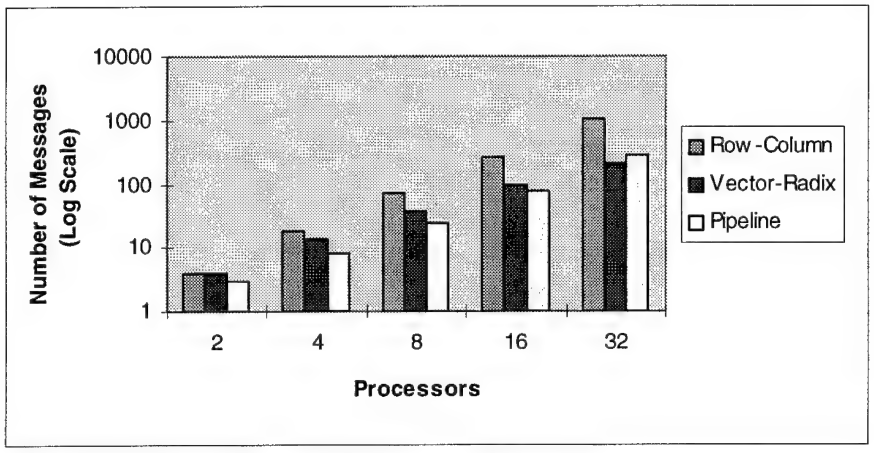


*Figure 3-17. Number of Messages Sent for the Three Implementations on a Logarithmic Scale.*

As can be seen in Figure 3-18, the pipeline sends the most data for all numbers of processors followed by the vector-radix and row-column, respectively. The gap between the three algorithms widens as the number of processors increases.

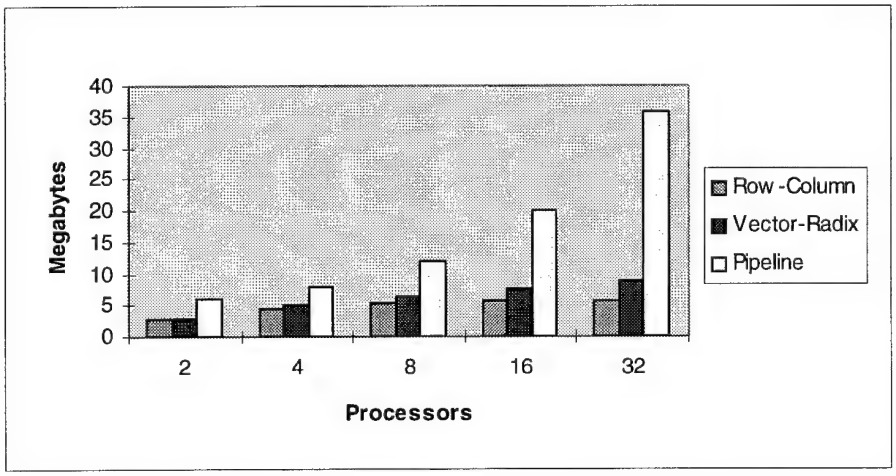


*Figure 3-18. Megabytes Sent by the Three Implementations.*

In summary, it appears that the row-column algorithm communication would well-suited for platforms which have low message startup time since it sends the most messages, but the least data. Also, it appears the pipeline would be the best for platforms

with high throughput because it sends the most data and the fewest messages. This analysis does not truly capture the effect of overlapping in the pipeline, however.

#### *3.2.3.4 Pipeline Complexity*

The runtime of the pipeline is not adequately described by its message traffic and computations. The pipeline attempts to maximize the overlap of communication as well as the overlap between computation of the row and column FFTs. The timing during the pipeline is taken when the pipe is full (steady state). The pipeline attempts to maximize throughput (images processed per unit time) versus the response time for a single image. Since these are different scenarios, a compromise must be made in order to compare the two implementations. The average time for a single FFT will be used as the measurement of the runtime of the pipeline. The average time for the FFT in the pipeline is given as the time from when the steady state starts until it ends, divided by the number of FFTs accomplished. The timing at the source/destination processor takes into account the sending of the image to the row processors and the receiving of the transformed image from the column processors. There is also a "gap time" as shown in Figure 3-19 which is the time the source/destination processor must wait between receiving and sending. If the communication time is *greater* than the FFT time, then the gap is close to zero, and the iteration time can be approximated by the time the source/destination processors to send an image to the row processors and receive one back from the column processors. In this case, the bottleneck is communication and the time of the FFT is hidden by the overlap of communication within the pipe. If the communication time is *smaller* than the FFT time,

the source/destination processor waits for the row or column processors, the gap time increases and the FFT becomes the bottleneck.

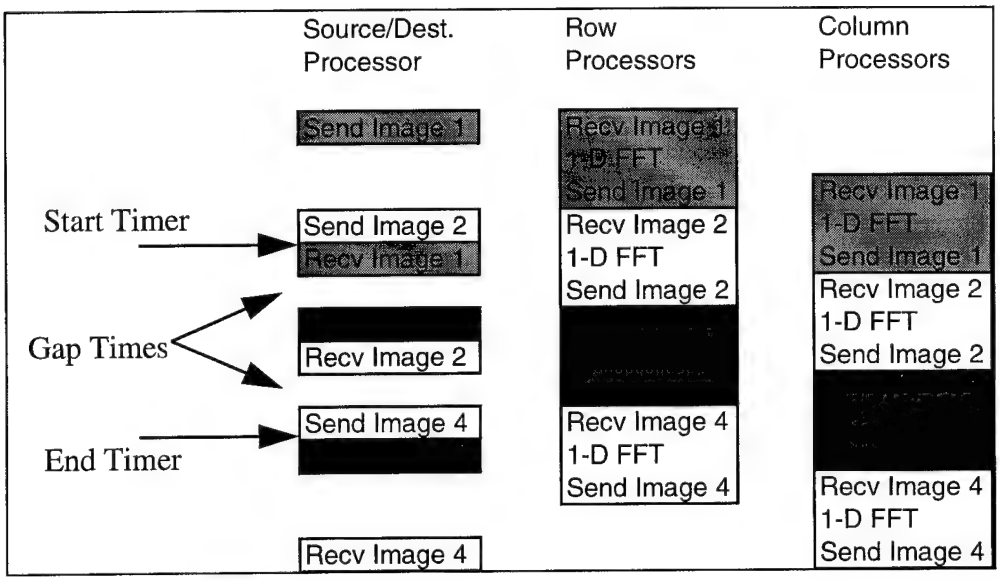


*Figure 3-19. Timing of Pipeline Implementation.*

### *3.2.3.5 Complexity Summary*

To summarize the discussion of 2-D FFT algorithm complexity, the following conclusions were made. First, the vector-radix algorithm is computationally more efficient than the row-column and pipeline algorithms. As the input size grows, however, there are diminishing returns because the number of computations grows at a slower rate. Second, the row-column sends the fewest messages, but the pipeline sends the most data and the differences grow as the number of processors increases. Third, the pipeline has the potential to hide the computation of the FFT by overlapping communication and computation, if computation time is bounded by communication time.

### *3.2.4 2-D FFT Code Optimizations*

This section provides a description of the techniques employed to improve the runtime of the parallel 2-D FFTs. It also details the optimizations used on each algorithm's implementation.

#### *3.2.4.1 Loop Invariant Code Removal and Common Sub-expression Elimination*

Both serial and parallel programs can benefit from two techniques known as loop invariant code removal (LICR) and common sub-expression elimination (CSE). LICR involves the moving of statements which are unaffected by loop variables outside of the loop to avoid their recomputation. CSE avoids recomputing a value by storing it in a temporary variable and replacing its computation with a reference to the new variable. These two techniques were combined quite frequently in all the programs to achieve some, albeit small, speedup. Most current compilers attempt to perform CSE and LICR, but any intervention from the programmer can help the compiler and increase the code's performance.

An example of these techniques is seen in the array indexing of the 2-D FFT code. To avoid statically declaring arrays when the image size is unknown until runtime, all 2-D arrays were declared using dynamic memory allocation. Normal 2-D array indexing (i.e. a[i][j]) could not be easily used because the memory is essentially a 1-D array. Therefore, the 2-D index calculation was done to properly access the array (i.e. a[i*n+j]).

The code segment in Figure 3-20 shows the index computation as well as the use of the two techniques described above.

| Unoptimized Code | Optimized Code |
|---|---|
| for (i=0; i<slice; i++)<br>  for (j=0; j<n; j++)<br>  b[j*n+taskid*slice +i] = a[i*n+taskid*slice+j]; | temp0 = taskid*slice;<br>for (i=0; i<slice; I++)<br>{<br>  temp1 = temp0+i;<br>  temp2 = i*n+temp0<br>  for (j=0; j<slice; j++)<br>  a[j*n+temp1] = b[temp2+j];<br>} |

*Figure 3-20. Loop Invariant Code Removal and Common Sub-expression Elimination..*

### *3.2.4.2 Derived Datatypes*

In each of the three implementations, communication is often preceded or followed by a re-ordering of data. For example, in the final two steps of the row-column implementation, the data is gathered to the destination processor, and then that processor serially transposes the entire matrix. It would be more efficient if the destination processor received the slices from the other processors and placed them in the matrix in transposed order rather than sequential order. In other words, point (i,j) is received into memory location (j,i). The MPI standard provides *derived datatypes* which allow data to be sent from or received into non-contiguous memory locations.[Snir96] The efficiency of these derived datatypes, however, is dependent on the MPI implementation's handling of them. Unnecessary copying or increased cache misses could worsen performance.

In the current MPICH implementation of MPI, derived datatypes cannot be used with collective communications such as MPI_Scatter, MPI_Gather and MPI_Alltoall.

The derived datatypes are larger in memory than the data they contain because of word boundary alignment. Collective communications use the size of the datatype to determine where consecutive data is to be placed, so the size mismatch causes data to be spaced improperly. To overcome this problem, the collective communication routines using derived datatypes were implemented by sends and receives instead of calls to the library functions. While this decreases the simplicity and readability of the code, these modified collective operations using derived datatypes offered better performance in most cases than the library routines which had added data copying.

#### *3.2.4.3 Packing*

Packing is a technique used to combine non-contiguous data into a single buffer. By completing only one send/receive operation instead of multiple sends/receives, the message startup overhead is avoided. The cost of packing is the copying of data into a buffer which can be expensive for large data. Derived datatypes should be used over packing when possible because derived datatypes avoid the extra copy into the buffer.

#### *3.2.4.4 Non-blocking (Asynchronous) Communication*

Non-blocking or asynchronous communication allows processors to overlap computation and communication by starting (posting) a send or receive, then doing some computation and then completing the send/receive. Asynchronous communication cannot be used if the computation is dependent on the results of the preceding communication, however. Also, it was not an improvement to replace a collective communication library

function with non-blocking sends and receives if no computation occurred in between the two.

#### *3.2.4.5 Row-Column Optimizations*

The row-column implementation uses derived datatypes and asynchronous communication to increase performance in two steps of the algorithm. In the matrix transposition step, after the first set of row FFTs, there are three subtasks which are accomplished (Sec 3.2.2.1). The rows are moved into "chunks", the chunks are transposed, then the AlltoAll communication takes place. The AlltoAll was implemented by each processor doing (p-1) MPI_Sendrecv's, which combine sending and receiving to avoid deadlock. The moving of data to the chunk data structure was avoided because the data was sent using a derived datatype and the data was received using a datatype which transposed the chunk while receiving. A non-blocking version of the code did not improve the performance of the communication because the MPI_Sendrecv is implemented with asynchronous communication at a lower level.

A second optimization was to replace the MPI_Gather and final, serial matrix transposition. At the destination processor, (p-1) non-blocking receives were posted followed by a transposition of the destination processor's slice (since sending to yourself can cause deadlock) and finally the receives were completed. The receives used a derived datatype to transpose the columns into rows as they were received.

A manual version of the MPI_Scatter library function was written, but it showed no improvement. Since the MPI_Scatter is broken down into non-blocking sends and receives, no speedup was expected.

#### *3.2.4.6 Vector-Radix Optimizations*

The vector-radix implementation used derived datatypes and packing to speed up communications in two steps of the algorithm. First, to avoid the bit-reverse of the rows of the input matrix, the source processor packed the appropriate rows into buffers and sent them as one large message. A derived datatype could not be used because the spacing between the rows destined for a processor is not constant, but is different based on the processor number and the total number of processors.

Second, to avoid the final permutation of rows by the destination processor after the gather, the MPI_Gather was replaced by sends and receives using derived datatypes. Again, non-blocking communication offered no better performance because no computation occurred between posting and completing the receives and the lower level functions use non-blocking communication.

#### *3.2.4.7 Pipeline Optimizations*

Like the row-column code, the pipeline used derived datatypes to avoid transposing data before or after communication. To avoid the chunk/unchunk process, the row processors used a chunk datatype to send data to the column processors. Also,

when the source/destination processor received from the column processors, the columns were transposed to rows as they were received to avoid the final, serial transposition.

### *3.3 Experimental Framework*

When comparing the performance of parallel routines, it is essential to develop a consistent framework which determines the degrees of freedom and, when possible, eliminates platform differences. This section discusses the parameters of the experiments as well as those factors which are different between platforms and must be taken into account during comparisons.

#### *3.3.1 Input Size*

The input data for all of the implementations were NxN matrices where N is a power of two. The implementations could be modified for non-square or non-power-of-two inputs, but the effort to accomplish this was not deemed necessary for this research. The size of N was varied from 128, 256, 512, 1024 to 2048. The exception to this was the Paragon which performed so poorly on the 2048 by 2048 size because the image occupies 32 megabytes of space which is half the main memory of a single node. These sizes were large enough to obtain meaningful results and limit the effects of program overhead, but small enough to fit in the main memory of the machines.

### *3.3.2 Number of Processors*

The number of processors for the parallel implementations was dependent on the number of processors available on each platform as well as the expected waiting time for results based on the platform's load. On the AFIT NOW, only 6 processors were available, so the row-column and vector-radix implementations were limited to 4 processors and the pipeline used a maximum of 5 processors. On the SP2 and Paragon, the number of processors was varied from 2, 4, 8, 16 and 32, with one extra for the pipeline code.

### *3.3.3 Messaging Layers*

As was discussed in Chapter II, the messaging layer used can have a large impact on communication performance. This section discusses the various messaging layers available for the different platforms.

#### *3.3.3.1 AFIT NOW*

The messaging layers for the AFIT NOW were chosen based on the availability of an MPI implementation for them, as well as their effectiveness. The first messaging layer used was the TCP/IP software from Myricom. This layer gave correct results and the MPICH implementation of MPI could be used with it. Because of the limitations of TCP/IP as discussed earlier, more efficient, experimental layers were investigated.

The University of Illinois's FM was the first experimental messaging layer installed and tested on the AFIT NOW. At first, FM would not even synchronize across more than two processors. Finally, the developers sent an executable compiled without optimization which allowed the synchronization to occur. At that point, FM appeared to work properly, but with slightly degraded performance. The authors of the FM software were unable to duplicate the synchronization problem because they were using a different platform than the AFIT NOW.

After problems with MPI-FM which are discussed later, the next experimental messaging layer installed on the AFIT NOW was Mississippi State's BDM. After many problems with the installation, BDM appeared to work correctly and with the expected performance. Again, however the MPI implementation had difficulties with correctness just as FM did.

#### *3.3.3.2 SP2*

On the SP2, there are two messaging layers available, TCP/IP and US. The TCP/IP messaging layer has the expected performance problems, so the more efficient US protocol was used. The advantage of US is that it is implemented in user space (hence the name) and does not require context switching for communication which is quite slow as discussed in section 2.2.4.

#### *3.3.3.3 Paragon*

The Paragon uses the native NX communication library. This library contains the necessary function calls to perform message passing on the Paragon. This library uses user space routines to avoid the context switching into supervisor mode. NX, in many ways, resembles the MPI communication constructs with both blocking and non-blocking communication supported.

### *3.3.4 Compilers and Compiler Optimizations*

The compiler effects the code schedule and size as well as the data access patterns and hence can strongly effect performance of programs. A higher optimization level can increase performance, but code size may grow because of techniques such as loop unrolling. Initially, it was intended to use the same compiler with the same optimization level on all three platforms, however it was not possible. On the AFIT NOW, the GNU C compiler, gcc, was used with an optimization level of -O3 (the highest). Unfortunately, on both the Paragon and SP2, MPI programs would not compile correctly with gcc and the system administrators were unable to help. On the Paragon, Intel's icc compiler was used with an optimization level of -O4. On the SP2, IBM's mpcc compiler was used with an optimization level of -O3. This variation in compilers is a possible point of deviation when comparing performance results across platforms.

### *3.3.5 MPI Implementations*

The MPI implementation determines how the MPI communication is performed at the underlying messaging layers. Unfortunately, the same MPI implementation could not be used on all platforms. This difference in MPI implementations is another possible factor in performance variations of identical code across platforms.

#### *3.3.5.1 AFIT NOW*

There are three MPI implementations available on the AFIT NOW with varying degrees of performance and correctness. The first MPI used was MPICH which uses TCP/IP as the messaging layer. MPICH can be used with any platform which supports TCP/IP and has also been ported to most MPPs.

The second MPI used was the University of Illinois's MPI-FM which uses the more efficient FM messaging layer. To simplify implementation, MPI-FM follows the MPICH ADI framework (described in Section 2.3.4) so only the 13 ADI functions would have to be implemented. Initially, MPI-FM seemed to work correctly even though its performance was somewhat disappointing. The more disturbing problem, however, was that as optimizations with derived datatypes were performed, MPI-FM gave incorrect results. Processors did not receive the proper data when receiving messages into a derived datatype and as a result, the 2-D FFT results were in error. Because of the performance and effectiveness problems with MPI-FM, it was not used as the primary MPI implementation in this research.

The third MPI implementation is MPI-BDM which is also an implementation of the MPICH framework, developed for Mississippi State's BDM messaging layer. MPI-BDM also had correctness problems like MPI-FM. When using collective operations, such as scatter, gather, and all-to-all, incorrect data was passed to the root processor. In other words, when processor 0 scattered the rows of an image, the other processors received the correct data, however the source processor, which only copies the data from one array to the other at the lowest level, received incorrect data. Since collective operations gave incorrect FFT results, the performance of any implementation using these routines is questionable. Luckily, in most of the optimized code, collective operations were replaced by sends and receives with derived datatypes, so the problem was avoided. For example, in the vector-radix and row-column optimized code, only the MPI_Scatter collective routine was used. The scatter was replaced by non-blocking sends and receives to achieve the same communication. The pipeline uses no collective operations since whenever a processors send data, it sends only to other processors and not itself.

#### *3.3.5.2 MPPs*

On the Paragon, the MPICH implementation was available which uses the efficient NX protocol for messaging, so it used the same MPI implementation as the AFIT NOW. Unfortunately, even though MPICH has been ported to the SP2, it was not available. The reason for this is that MPICH uses the rlogin command to spawn remote processes and, for security reasons, it has been disabled on the MSRC SP2. The MSRC SP2 uses IBM's implementation of MPI which does not use the MPICH framework. This

inconsistency of MPIs between the platforms adds a degree of uncertainty in the performance comparison.

### *3.3.6 Number of Iterations/Runs*

The reported results in this thesis are for two runs of each program, with each run performing 5 iterations of the FFT. Performing two separate runs of the program eliminated any transient problems with operating system events such as context switches. The first iteration of each run is thrown out to remove the effects of loop overhead and page faults. For example, a 1024x1024 row-column FFT on the Paragon with 32 processors is 63% slower on the first loop as compared to the average of the other iterations. Therefore, there were eight samples per trial from the two runs of four iterations.

This number of iterations was large enough to ensure a small variance. For example, on the row-column with a 512x512 input size with 4 processors, the variance of the average runtime was 0.030% on the Paragon, 0.076% on the SP2, and 0.013% on the AFIT NOW. Using the highest variance (SP2), a precision of .1% was obtained with a 99% confidence interval. This precision is sufficient for this research because only a rough measure of the relative performance of the platforms is necessary to make qualitative judgments about their performance. On the MPPs, there was exclusive access to the compute nodes, so there was no contention for the processor from other jobs which kept times consistent. On the AFIT NOW, all tests were run at times that there were no interactive users, in order to minimize context switching to service user actions.

# *IV. Analysis*

Chapter II explained the differences between NOWs and MPPs and looked at the specific platforms used in this research. Chapter III gave the background of the 2-D FFT application, a description of the code developed, and outlined the framework for the performance comparisons of the code on different platforms. The purpose of this chapter is to present the results of the experiments conducted and give explanations for performance differences. Trends within the data are analyzed in order to determine the scalability of algorithms as well as the suitability of the different platforms for this application.

This chapter is organized as follows. First, the three platforms are benchmarked for computational and communication performance. Second, the three algorithms are analyzed and compared on each platform. Each code's performance is detailed including optimization improvements, processor scalability, and problem size scalability. Also, the codes are compared to determine the best algorithm for a given problem size and number of processors on the platform. Third, the algorithms are compared across the three platforms. Each code's performance is analyzed for runtime and speedup across all the platforms. Fourth, the best code for each platform is compared between the platforms to determine the suitability of the platforms for the 2-D FFT application.

## *4.1 Benchmark Results and Metric Comparisons*

As was discussed in Chapter III, parallel program performance can be broken into two parts, computation and communication. The computation time depends mostly on

the processor, in conjunction with the compiler and memory hierarchy. The communication time depends on the processor, the network interface, the network, and, at the software level, the messaging layers and communication library.

### *4.1.1 Computation Benchmark*

To better understand the computational differences between the different platforms, serial 2-D FFTs were performed. The resultant runtimes were also used for the speedup (see Sec 4.1.3) computations for the parallel algorithms. There were four different serial 2-D FFT implementations used on the three platforms. The first was the basic row-column which is a serial version of the row-column described in Chapter III. The basic procedure is: row FFTs, matrix transpose, column FFTs, matrix transpose. The basic row-column came with a slow 1-D FFT. The second implementation used the Japanese split-radix 1-D FFT for better FFT performance. The third implementation was the 2-D FFT from the University of Tokyo, Japan.[Ooura97] This uses the row-column method with their fast 1-D FFT, however the transpose is performed differently than the basic row-column. Instead of transposing the entire matrix, each column is copied to a temporary row, the FFT is performed, and then the column is copied back. The fourth implementation is the serial vector-radix, from which the parallel vector-radix was developed. [Pitas93]

The performance ranking of the four serial 2-D FFTs was consistent across all the platforms. The row-column with the Japanese 1-D FFT was the best, followed by the

Japanese 2-D, the vector-radix and finally the basic row-column with the slow 1-D FFT. Figure 4-1 shows the performance of the four implementations on the 200 MHz Sparc Ultra. The runtime is not shown, because of the large range of values. Instead, a metric called megaflops (million floating point operations) is used. For a given input size, megaflops are given by $(10N^2\log_2 N/\text{runtime})/10^6$. This assumes that there are 5 complex operations (multiplies and adds) per point in the 1-D FFT. The megaflops rating represents an input-scaled runtime and approximates the rate of computation for a given problem size. This definition of megaflops is different from the one often used in the literature. Some researchers count the actual number of floating point operations performed by the CPU. This approach is flawed because it doesn't take into account the time taken for the operations which varies depending on a hardware or software implementation and which may vary from platform to platform.

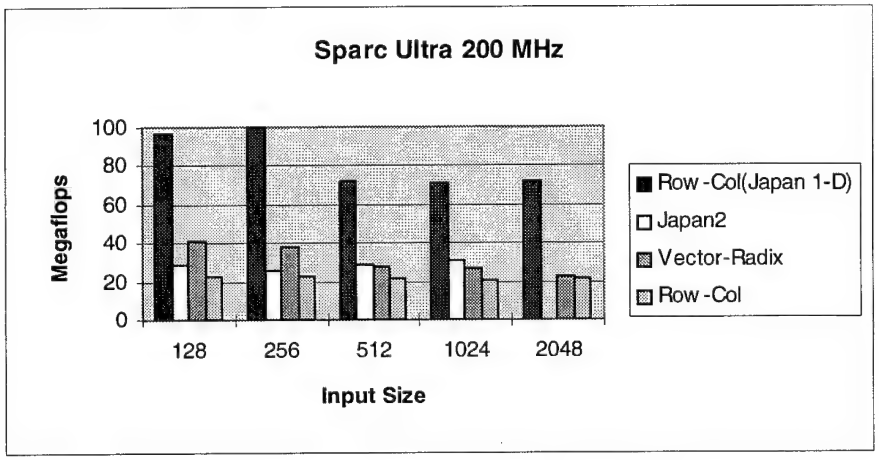


*Figure 4-1. Average Megaflops of Four Serial 2-D FFTs on a 200 MHz Ultra Sparc.*

The performance of the serial 2-D FFTs on the other platforms is not detailed, but merely summarized. The row-column with the Japanese split-radix 1-D FFT was the best

on all three platforms. Between the row-column and the Japanese code, there was a 77-71% difference on the Sparc Ultra 200, 30-40% difference on the SP2, and less than 10% difference on the Paragon. On the Ultra 200 and SP2, the vector-radix was better than the Japanese code for small data sizes (128x128).

To show the computational differences between the platforms, the best serial 2-D was compared across platforms. Figure 4-2 shows the performance of the row-column with the Japanese 1-D FFT for the three platforms (in megaflops).

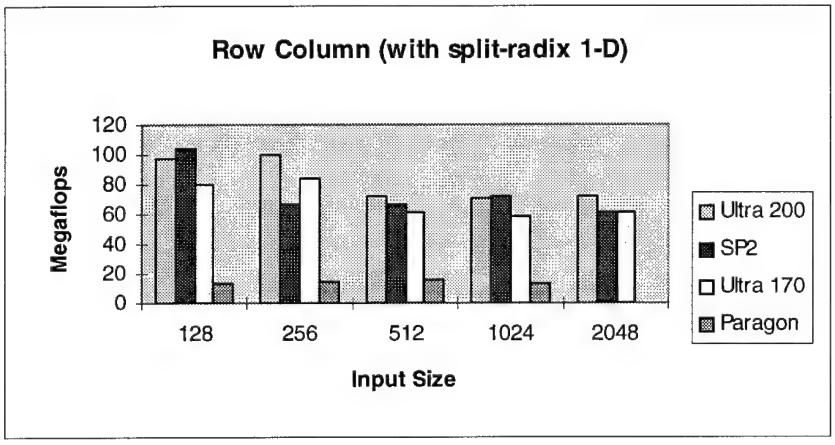


*Figure 4-2. Average Megaflops of Serial Row-Column 2-D FFT for Each Platform*

From the serial results, it appears that the SP2 and Ultra 200 are approximately equal. There is definitely no clear-cut winner between the two platforms. The Paragon is by far the worst of the three with one-third to one-fifth of the performance of the SP2 or Ultra 200. It was expected that the Paragon would perform the worst computationally since it employs a processor which uses early 1990's technology. Also, the Paragon has only a 32 KB data cache and none of these problem sizes fit completely in the cache.

What was surprising was that the SP2 was able to meet and sometimes exceed the performance of the Ultra 200. As discussed in Chapter II, one of the main advantages of NOWs is better computational performance compared to MPPs. The SP2's use of commodity parts, specifically the RS/6000 line of microprocessors, which is also used in their workstations, allows them to upgrade the processor with little redesign of the SP2 network connection. As a result, the Power2 processor is competitive with the Ultra 200, even beating it on the Spec 95 floating point benchmark suite (15.4 vs. 9.8 Spec FP).[Spec95]

### *4.1.2 Communication Benchmark*

To quantify the communication performance of the three platforms, bandwidth and latency tests were performed on each platform using an original MPI benchmark program. This test determines the one-way communication time from one processor to another by halving the round-trip time for a message to travel from the source to the destination and back. The first test measured the latency for small message sizes from 0 to 32K bytes at increments of 256 bytes. Figure 4-3 shows the latency (in msec) for the message sizes. The three "user space" messaging layers perform nearly the same, with the Myrinet TCP/IP the worst by far. This data supports the discussion in Section 2.2.4 about the inadequacy of TCP/IP for high performance communication.

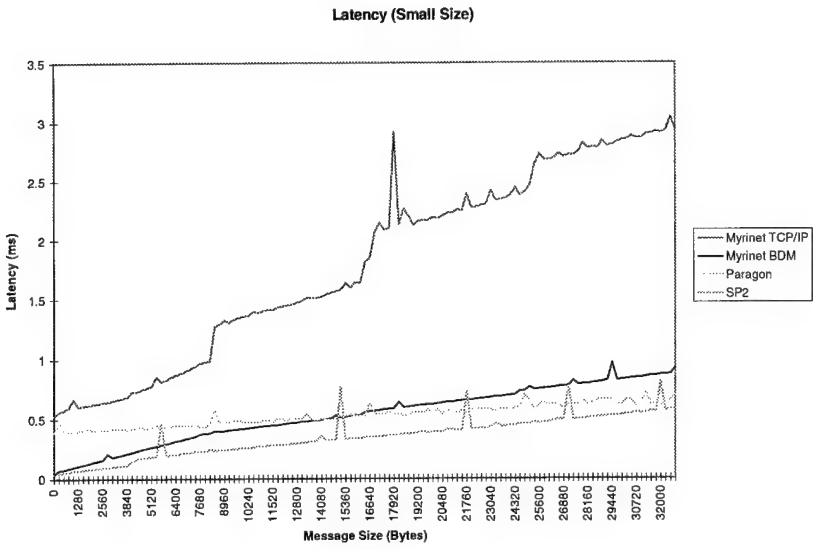


*Figure 4-3. Message Latency for Each Platform for Small Message Sizes.*

To approximate the message startup cost on each platform, the 0-byte latency is used. This data point measures the time for the node to create an empty message, send it and receive it. Table 4-1 shows the 0-byte latency and message startup time for the three platforms. The startup time shown at the 0 byte point in Figure 4-3 is divided by two since both the send and receive are included in the latency. When message sizes are relatively small, startup time becomes a factor in communication time since the relative time of message startup to message transfer time is closer. For larger messages the message transfer time is greater, so message startup is less of a factor. The exact message size that transfer time becomes dominant depends on the relative values of message startup and transfer time.

*Table 4-1. 0-Byte Latency and Message Startup Time for Each Platform.*

| Platform | 0-Byte Latency (msec) | Startup Time (microsec) |
|---|---|---|
| SP2 | 0.0361 | 18.05 |
| Paragon | 0.384 | 192.2 |
| Myrinet (BDM) | 0.040 | 20.0 |
| Myrinet (TCP) | 0.531 | 265.5 |

Another important test was to benchmark the throughput (MB/sec) for large message sizes. The message size was varied from 0 to 4 MB at 16 KB intervals. This range encompasses almost the entire range of message sizes sent in an FFT. For example, when scattering the image to the processors in the row-column or vector-radix algorithms, messages of size (N/p)*N*8 (assuming single precision FP) are sent. For N=1024 and p=8, this equates to 1 MB of data. Figure 4-4 shows the results of the throughput test for the three platforms.

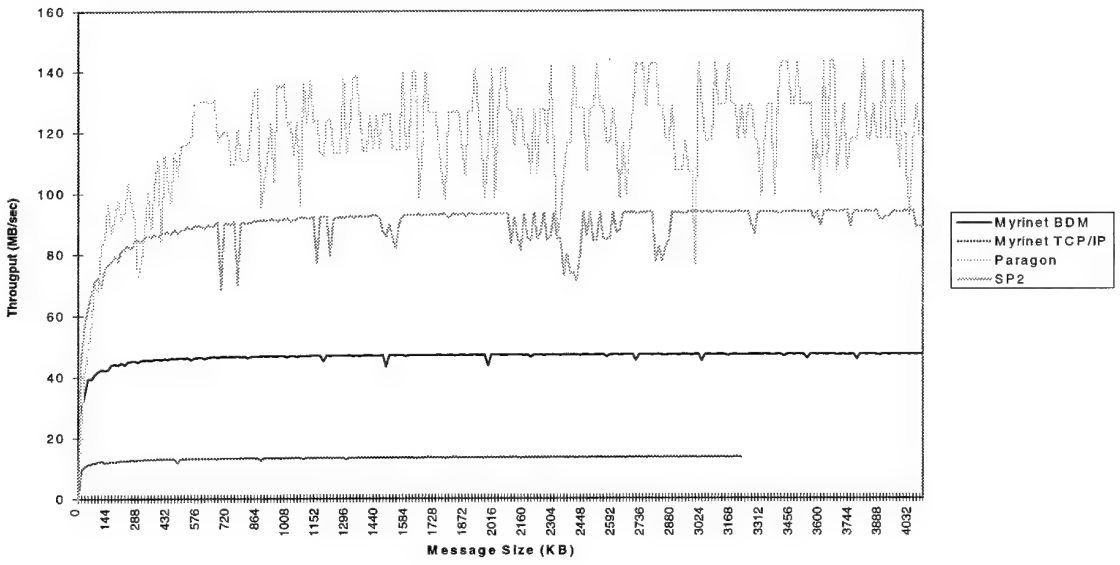


*Figure 4-4. Measured Throughput of Each Platform for Large Message Sizes.*

A few important conclusions can be drawn from the data in Figure 4-4. First, despite being the oldest of the three platforms, the Paragon shows the best throughput by far, primarily because its NIC is connected to the cache coherent memory bus. Second, the faster the communication performance, the greater variability in throughput, as seen by the spikes in the throughput curves. This variability is probably attributed to the variability in message transfer across the memory or I/O bus. Third, the messaging layer can have a huge impact on performance as seen by the difference between Myrinet with BDM and TCP/IP. Fourth, maximum bandwidth (sometimes called hardware or theoretical bandwidth) is a poor measure of actual performance. Table 4-2 illustrates this point by showing the maximum one-way bandwidth and the measured throughput as seen by this benchmark.

*Table 4-2. Maximum Theoretical Bandwidth and Measured Throughput of Each Platform.*

| Platform | Max. Theoretical Bandwidth (MB/sec) | Max. Application Throughput (MB/sec) |
|---|---|---|
| ***Paragon*** | 200 | 122 |
| ***SP2*** | 150 | 91 |
| ***Myrinet (BDM)*** | 80 | 47 |
| ***Myrinet (TCP)*** | 80 | 13 |

In summary, the Paragon exhibits the best throughput, followed by the SP2 and AFIT NOW, but has the largest message startup time. The SP2 has twice the throughput of the AFIT NOW, but they have approximately the same message startup time. The dominant part of communication time, however, is dependent the values of $t_s$ and $t_w$ for a given platform, as well as the size of the messages being sent. Table 4-3 shows the startup time and message transfer time for a (N/p)*N*8 byte message (scatter and gather) with four processors. With the exception of the Paragon, the message transfer time dominates the message startup time, even at the smallest input size. As input sizes increase, this difference increases, making the number of bytes sent a more important factor in communication time than the number of messages sent.

*Table 4-3. Message Startup Time and Message Transfer Time for the Three Platforms on a Scatter or Gather Operation with 4 Processors and Varying Input Sizes.*

| Platform | Startup Time (msec) | Message Transfer Time(msec) | | | | |
|---|---|---|---|---|---|---|
| | | 128 | 256 | 512 | 1024 | 2048 |
| Paragon | 0.384 | 0.256 | 1.024 | 4.096 | 16.384 | 65.536 |
| SP2 | 0.0361 | 0.343 | 1.372 | 5.488 | 21.952 | 87.808 |
| AFIT NOW (BDM) | 0.04 | 0.665 | 2.66 | 10.64 | 42.56 | 170.24 |
| AFIT NOW (TCP/IP) | 0.531 | 2.4 | 9.6 | 38.4 | 153.6 | 614.4 |

### *4.1.3 Metrics*

When comparing the vector-radix, row-column and pipeline implementations on the SP2, Paragon, and AFIT NOW, it is important to use a metric which is equitable to the different algorithms and different platforms, and also is applicable to the comparison being made. There are three primary metrics for comparing parallel algorithm performance: runtime, speedup and efficiency. Runtime is simply the time from the start to finish (possibly ignoring program start or first-loop computations). All things being equal, runtime is the best metric because it measures the time to complete a program. Also, when comparing platforms to use or purchase, runtime is the best measure since the time to finish a program is the bottom line that the end-user observes.

The second metric, speedup (S) is defined as the best serial runtime ($T_s$) divided by the parallel runtime ($T_p$), for a given problem size.

$$Speedup(S) = \frac{Serial\ Runtime}{Parallel\ Runtime} = \frac{T_s}{T_p} \quad (7)$$

Speedup determines the improvement of parallelizing a problem on a platform. If speedup is greater than one, then parallelization is an improvement, otherwise the serial program is faster and therefore preferable. Theoretically, the speedup cannot be greater than the number of processors (p). If speedup is greater than the number of processors, it is called superlinear. This occurs only if a sub-optimal serial program is used, or the effects of the memory hierarchy cause smaller, parallel problem sizes to fit in cache or main memory, while the serial problem does not.[Kumar94] Speedup is also useful when assessing scalability which is discussed in the next section.

The third metric is efficiency which is defined as the speedup divided by the number of processors.

$$Efficiency(E) = \frac{Speedup}{Processors} = \frac{S}{p} = \frac{T_s}{pT_p} \quad (8)$$

Efficiency measures the improvement obtained by adding more processors. If efficiency drops dramatically as processors are added, then there are diminishing returns to adding more processors, typically because communication costs rise. Theoretically, efficiency should never be greater than one because speedup cannot be greater than the number of processors.

Although runtime is generally the best metric for performance comparisons, it may be difficult to employ fairly when comparing different algorithms on the same platform. For the three algorithms used in this research, runtime may not be a fair metric because the pipeline has the advantage of an extra processor. For example, it would be analogous to comparing the vector-radix with 8 processors to the row-column with 2 processors. The row-column with 2 processors is 37% slower than the 8 processor vector-radix on the Paragon, however the row-column with 8 processors is approximately 33% *faster*. This somewhat contrived example illustrates the point that the correct metric must be used. For the case of different numbers of processors, efficiency can be used as an additional metric to runtime because it takes into account the number of processors used.

### *4.1.4 Scalability*

Scalability is a *qualitative* term used to describe parallel platforms. Because it's definition is subjective, vendors use whatever criteria that make their platform look the best. It may be the most over-used, misunderstood term in parallel processing. For this research, two simple measures of scalability are used: processor scalability and problem size scalability. Processor scalability is the ability of an algorithm to use additional processors and continue to increase performance. If speedup increases as processors are added, then a code can be called processor scaleable.

The second scalability metric is problem size scalability. Problem size scalability measures the performance of an algorithm as problem size is increased. If speedup increases as the problem size is increased, then the algorithm is considered problem size scaleable. An algorithm's speedup may peak at a given problem size or may vary with the ratio of problem size to processors (N/p).

## *4.2 Paragon Results*

The purpose of this section is to detail the performance of the Paragon for the three algorithms separately and then compare their performance to determine the best code for a given number of processors and input size. The performance results on the Paragon were the most regular of the three platforms. Scalability and performance trends were easily discovered and all the optimizations performed as expected.

### *4.2.1 Vector-Radix*

The vector-radix performed the worst of the three implementations on the Paragon; however it exhibited some good performance trends. Figure 4-5 shows the time breakdown of the different sections of a vector-radix with four processors as the *input size* increases. The runtimes are scaled to equal sizes to show the percentage of total runtime of the five sections of the code. This figure shows that the two computational

sections of code, the FFT computation (without communication) and the bit-reverse, take the majority of the runtime (50-70%) for the four processor case.

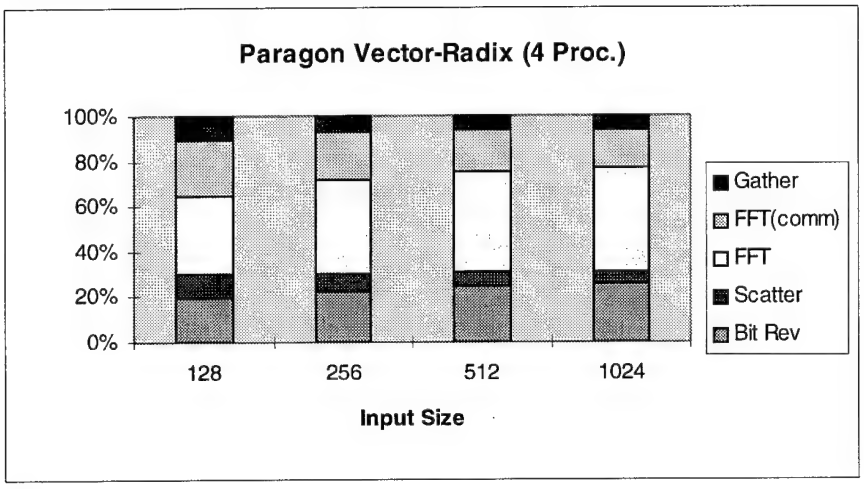


*Figure 4-5. Time Breakdown of Vector-Radix with 4 Processors on the Paragon.*

Figure 4-6 shows the time breakdown for the 512x512 problem size as *processors* are increased. The communication portions increase their percentage of the runtime, but the bit-reverse becomes the most dominant, taking almost 50% of the runtime for the 32 processor case. Since the bit-reverse is done serially by the source processor, this limits the speedup because that step cannot be further parallelized.

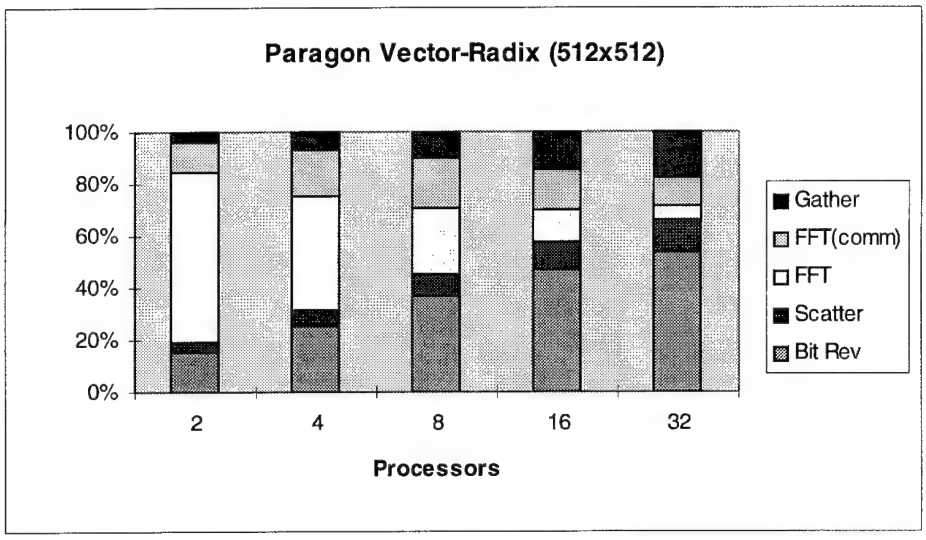


*Figure 4-6. Time Breakdown of Vector-Radix for 512x512 Input Size on the Paragon.*

#### *4.2.1.1 Optimizations*

The optimizations of the vector-radix all proved worthwhile and reduced average total runtime 50% to 70% over the unoptimized version of the code, depending on the number of processors. There were three optimizations performed as discussed in Section 3.2.4.6: packing of rows to avoid bit-reversing along the rows, loop invariant code removal (LICR) and common sub-expression elimination (CSE) for the FFT computations, and a derived datatype to avoid the final permutation of the image. The performance of the optimizations varied mainly with the number of processors and not the input size. Table 4-4 shows the average decrease in runtime for each optimization as the number of processors increases as well as the total decrease in runtime.

*Table 4-4. Percentage Decrease in Runtime of Vector-Radix Optimizations on the Paragon.*

| Processors | Bit Rev/ Scatter | LICR/ CSE | Gather | Total Runtime |
|---|---|---|---|---|
| **2** | -21.81% | -33.46% | -94.01% | **-51.91%** |
| **4** | -23.85% | -31.03% | -93.12% | **-57.72%** |
| **8** | -27.40% | -32.68% | -92.77% | **-65.01%** |
| **16** | -28.37% | -30.82% | -92.01% | **-68.71%** |
| **32** | -28.22% | -29.37% | -90.81% | **-69.99%** |

The performance of the bit-reverse optimization increases as the number of processors increases, making it very scaleable. The other two optimizations lose performance only slightly as the number of processors increase making them also scaleable. Since the bit-reverse becomes a larger percentage of the total runtime of the vector-radix as processors increase, the overall runtime continues to improve as processors are added which makes the optimized code a clear, scaleable improvement.

#### *4.2.1.2 Scalability*

To analyze the scalability, it is useful to see how the speedup changes as the number of processors and the input size change. Figure 4-7 shows the speedup curves for the four input sizes as the number of processors increases. As the number of processors increases the speedup always increases, making adding more processors always an improvement, at least to 32 processors.

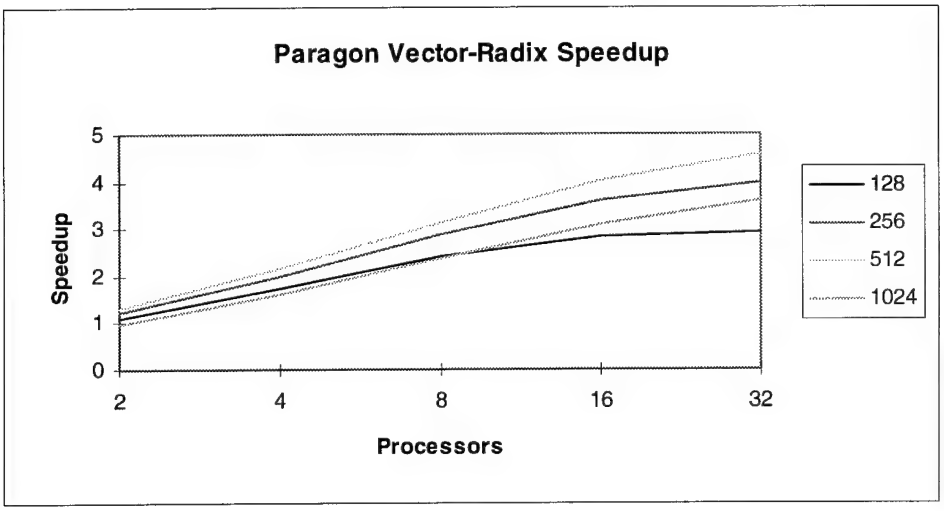


*Figure 4-7. Speedup of Vector-Radix on the Paragon*

### *4.2.2 Row-Column*

The row-column algorithm was second in performance on the Paragon ahead of the vector-radix. Figure 4-8 shows the time breakdown of the different sections of the algorithm for four processors with varying input size. As in the vector-radix, the FFT computation is a major part of the runtime (40%), but unlike the vector-radix, the gather operation becomes more dominant as the problem size increases. The gather time increases with problem size because, during the gather step, the matrix is being transposed. Despite an optimization to overlap the transpose with the communication, the cache effects become evident. This situation is discussed further with the gather optimization description in Section 4.2.2.1.

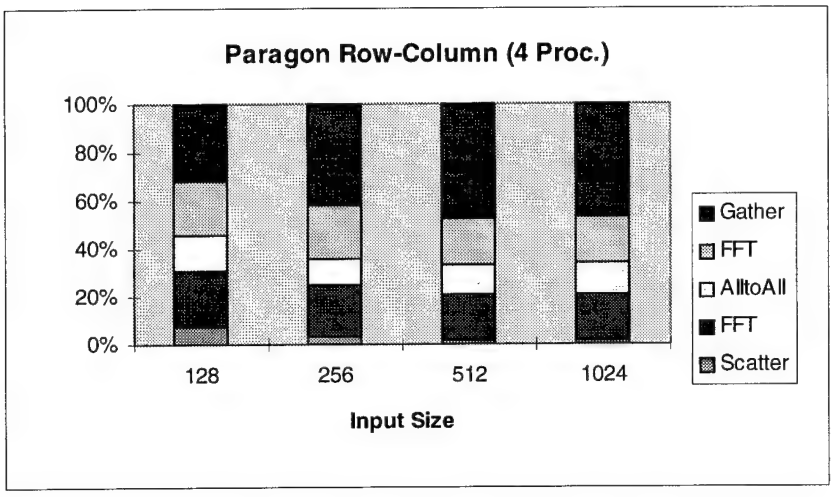


*Figure 4-8. Time Breakdown of Row-Column with 4 Processors on the Paragon*

Figure 4-9 shows the time breakdown for a 512x512 row-column for varying numbers of processors. As with problem size, the gather increases its percentage of runtime dramatically as the number of processors increases, reaching 80% at the 32 processor case. This occurs because as the number of processors increases, the time of the other components of the runtime decrease, while the gather remains approximately the same. As is discussed later, this is mainly due to the cache misses caused by placing data in non-contiguous locations.

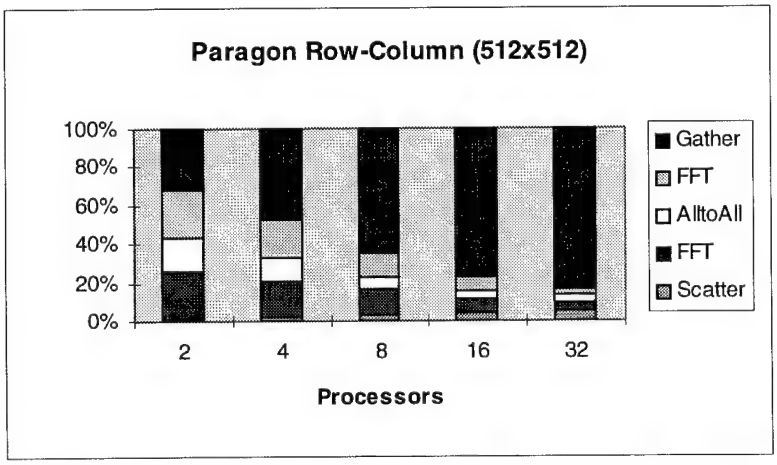


*Figure 4-9. Time Breakdown of the Row-Column for 512x512 Input Size on the Paragon.*

### *4.2.2.1 Optimizations*

The optimizations on the row-column code also performed well as in the vector-radix. There were two optimizations employed as described in Sec 3.2.4.5. First, the AlltoAll step, which transposes the entire matrix between row and column FFTs, was re-written with derived datatypes to avoid extra copying of data to a separate "chunk" data structure. This optimization performed better as the ratio of input size to processors (N/p) increased. Figure 4-10 shows the percentage decrease in runtime for the AlltoAll section. Clearly, as N/p (the number of rows of each processor's image) increases to 16 and greater, the optimization is an improvement.

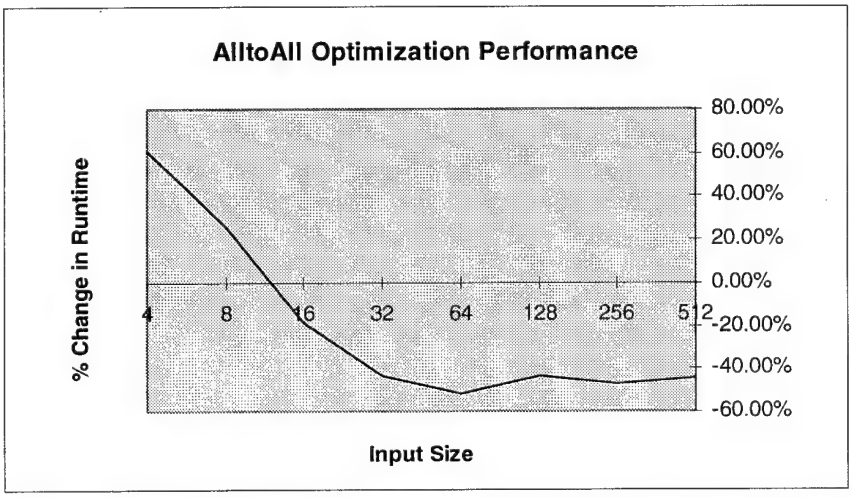


*Figure 4-10. Percent Change in Runtime as N/p Varies for AlltoAll Optimizations in Row-Column on the Paragon.*

The second optimization was a change of the final gather operation to use derived datatypes to avoid the final, serial transpose. Table 4-5 shows the range of improvement for each input size as the number of processors increases. It appears that this optimization would not scale well to larger input sizes since the improvement decreases

significantly from 128x128 to 1024x1024 points. It can be concluded that the derived datatype is still subject to the performance degradation with problem size, just as a serial transpose is. Sequential memory access increases locality and hence cache hits. When transposing, the processor writes to memory locations which are strided by N points (8N bytes) in a column-major fashion. As N increases, cache misses increase, and the runtime of the transpose increases. Since there is a decrease in runtime, however, the cache misses of the serial transpose are overlapped with the communication of the gather operation, making the derived datatype an improvement.

*Table 4-5. Percentage Change in Runtime for Gather Optimization in Row-Column Code on the Paragon.*

| Input Size | % Decrease in Runtime |
|---|---|
| 128 | -60.20% |
| 256 | -53.01% |
| 512 | -39.96% |
| 1024 | -39.05% |

#### *4.2.2.2 Scalability*

The processor and problem size scalability of the optimized row-column code have positive trends on the Paragon. Figure 4-11 shows the speedup curves for the four input sizes. The processor scalability is good because the speedup generally increases as processors are added. Even though the 256x256 input size shows the highest speedup,

the 512x512 and 1024x1024 continue to show a positive speedup trend, so the row-column can be considered problem size scaleable.

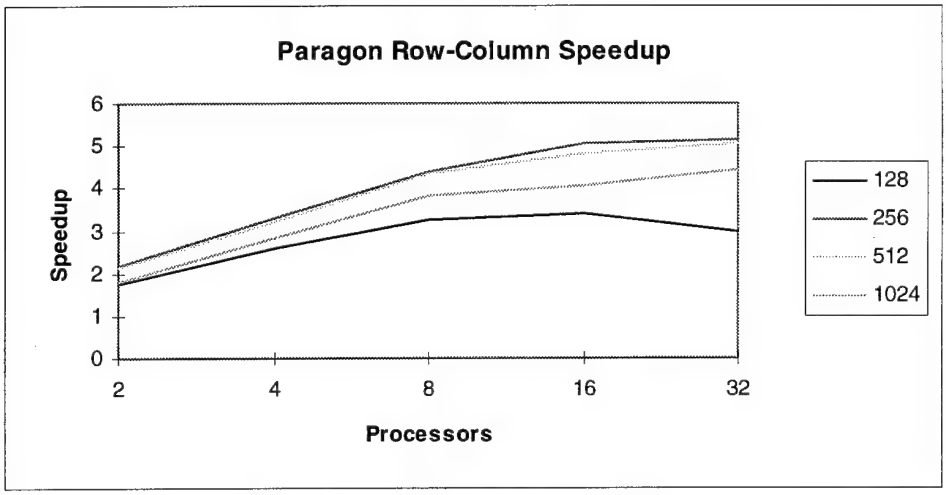


*Figure 4-11. Speedup of Row-Column on the Paragon.*

### *4.2.3 Pipeline*

The pipeline performed the best of the algorithms when considering runtime only. In many ways it exhibited similar trends to the row-column, which is not unexpected since they do the exact same computations with a different data distribution and communication pattern. Also, similar optimizations were performed on both to avoid extra copying of data.

#### *4.2.3.1 Optimizations*

There were optimizations used in the original pipeline code (derived datatypes and asynchronous communication), but no baseline code without these optimizations was written. Since the performance of the row-column optimizations on the AFIT NOW and Paragon proved to be worthwhile, time was not taken to write unoptimized code.

#### *4.2.3.2 Scalability*

The pipeline exhibits some scalability as seen in Figure 4-12. For all input sizes, the processor scalability drops at 9 processors. This occurs as the performance of optimizations with derived datatypes lessen, just as with the row-column code on the Paragon (Sec 4.2.2). There is always speedup, making the adding of additional processors beneficial, but the benefits decrease beyond 9 processors such that the pipeline is slower with less processors. A possible explanation for this peak is the exponential increase in the number of bytes sent as seen in Section 3.2.3.3. As the data sent increases, it reaches a point where the pipeline loses performance due to the communication overhead.

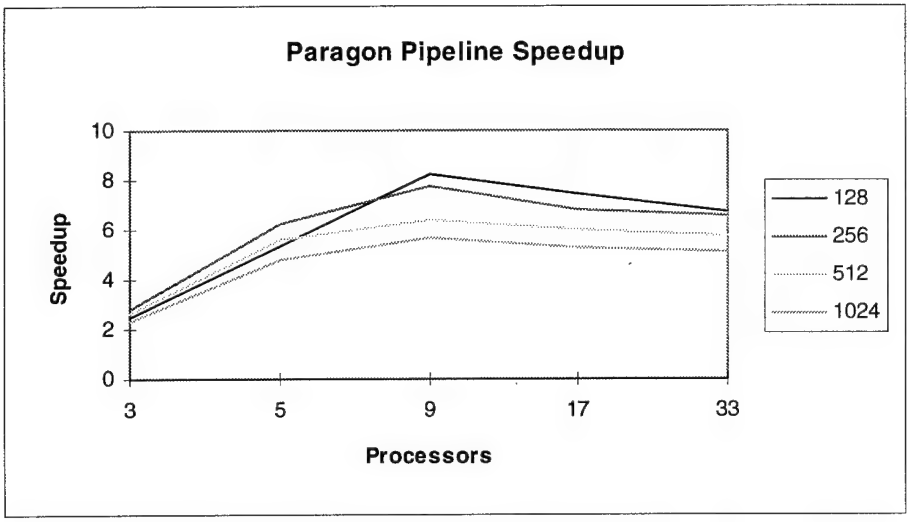


*Figure 4-12. Speedup of Pipeline on the Paragon.*

Although it may not be obvious from Figure 4-12, the pipeline exhibited superlinear speedup with 5 processors for data sizes less than 1024x1024. As discussed in Section 4.1.3, superlinear speedup can occur for two reasons: a suboptimal serial algorithm or memory hierarchy effects. In this case it does not occur because of a suboptimal serial algorithm because four different serial 2-D FFTs were tested for the best performance. Also, the best serial 2-D FFT used the same 1-D FFT that the pipeline used. A more logical explanation is the overlapping of communication and computation in the pipeline. Since the Paragon has a slow processor, the overlapping of communication and computation can hide the computation time of the FFT which is a bottleneck.

### *4.2.4 Row-Column vs. Vector-Radix vs. Pipeline*

To determine the best algorithm for a given input size and number of processors on the Paragon, the row-column, vector-radix and pipeline performance were compared.

When speedup is used as the metric, the pipeline is by far the best of the three algorithms. Figure 4-13 shows the speedup of the three algorithms for the 1024x1024 input size as the number of processors increases. All the input sizes showed approximately the same relative performance, so only the 1024x1024 case is shown. Although the pipeline performed the best, it's speedup decreased beyond 9 processors, while the other two algorithms always benefited from more processors. Since the pipeline lacks processor scalability, it is conceivable that the row-column or vector-radix could surpass it in speedup if the number of processors is increased beyond 32.

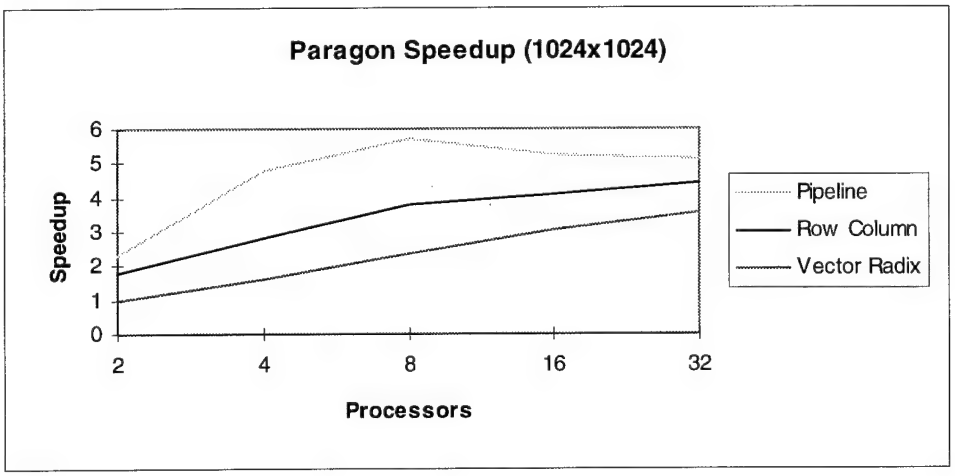


*Figure 4-13. Speedup of the Three Algorithms for 1024x1024 Input Size on the Paragon.*

A better metric for comparison of the three algorithms is efficiency, as discussed in Section 4.1.3. The efficiency data results in different, more fair conclusions because the pipeline uses more processors. Figure 4-14 shows efficiency curves for the row-column, vector-radix and pipeline algorithms for the 1024x1024 input size. The pipeline has the best efficiency, except in the 2/3 processor case, where the row-column is slightly

better (~15%). Again, as the number of processors increases beyond 8/9, the difference between the algorithms lessens.

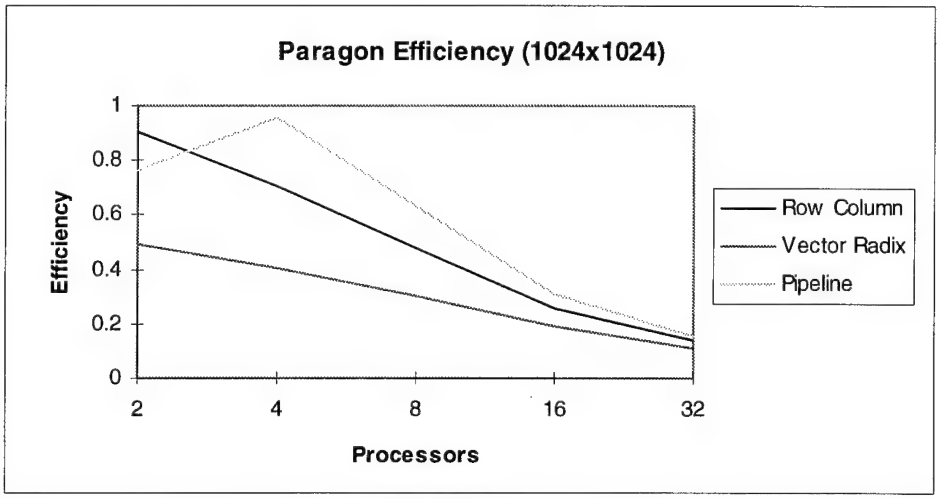


*Figure 4-14. Efficiency of the Three Algorithms for 1024x1024 Input Size on the Paragon.*

The reason for the difference at 2/3 processors is explained by analyzing the communication times and the efficiency equation. The same number of computations is performed by both algorithms since they use the same 1-D FFT, so computation time is approximately equal. When communicating at the 2/3 processor case, however, the pipeline sends 3 messages, and the row-column sends 4. Since message startup time is the high on the Paragon compared to its throughput (Sec. 4.1.2), this difference is significant. When calculating efficiency, the speedup is divided by the number of processors. At the 2/3 processor case, the row-column uses 33% less processors and sends 33% more messages. However, at the 4/5 processor case, the row-column uses 20% less processors, but sends 75% more messages. This difference grows as the

number of processors increases past 2/3, and the pipeline outperforms the row-column in terms of efficiency.

Other advantages of the pipeline are computation/communication overlap and congestion avoidance. As discussed in Sec 3.2.3.4, the pipeline can "hide" the FFT computation. This is especially beneficial for the Paragon which has better communication than computational performance, as compared to other platforms. The pipeline also avoids congestion, which is important in a direct network such as the Paragon. When two non-contiguous processors communicate across the mesh, the packets must travel on interconnection links, on which other processors are also possibly communicating. During the AlltoAll step of the row-column, each processor must swap with all other processors which ensures that each link is being utilized and therefore congestion is unavoidable. The pipeline avoids some of this congestion because instead of swapping data, the row processors only send data to the column processors, which means larger messages are sent, but less congestion results.

### *4.3 SP2 Results*

The purpose of this section is to detail the performance of the SP2 for the three algorithms separately and then compare their performance to determine the best code. The SP2 yielded similar performance trends compared to the Paragon, except in the area of code optimizations.

### *4.3.1 Vector-Radix*

The vector-radix performed the worst of the three algorithms, as it did on the Paragon. Figure 4-15 shows the time breakdown of the vector-radix with four processors. Ignoring the 128x128 case, all the steps of the code decrease their percentage of the total runtime, except the FFT computation (without communication). The bit-reverse is dominant for 256x256 points, but by 2048x2048 points, it is equal with the FFT computation. As with the Paragon, speedup is limited by the serial bit-reverse performed by the source processor. The bit-reverse increases its portion of runtime significantly from 128x128 to 256x256 because the image size exceeds the SP2's data cache (128 KB).

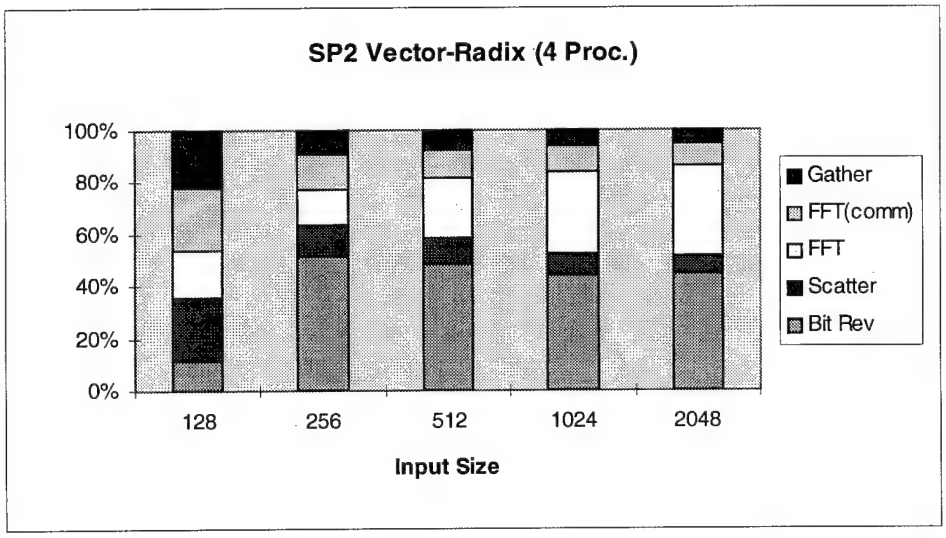


*Figure 4-15. Time Breakdown of Vector-Radix with 4 Processors on the SP2.*

Figure 4-16 shows the time breakdown of a 512x512 point vector-radix as processors are increased. The FFT portions of the runtime decrease to almost nothing as the number of processors increase, while the gather, scatter and bit-reverse increase their percentage of total runtime. The scatter becomes a greater portion of runtime as the

number of processors increases because the rows must first be manipulated before being sent, which increases the overhead in sending messages. As the number of processors and hence the number of messages increases, this send overhead causes the scatter operation to dominate the runtime.

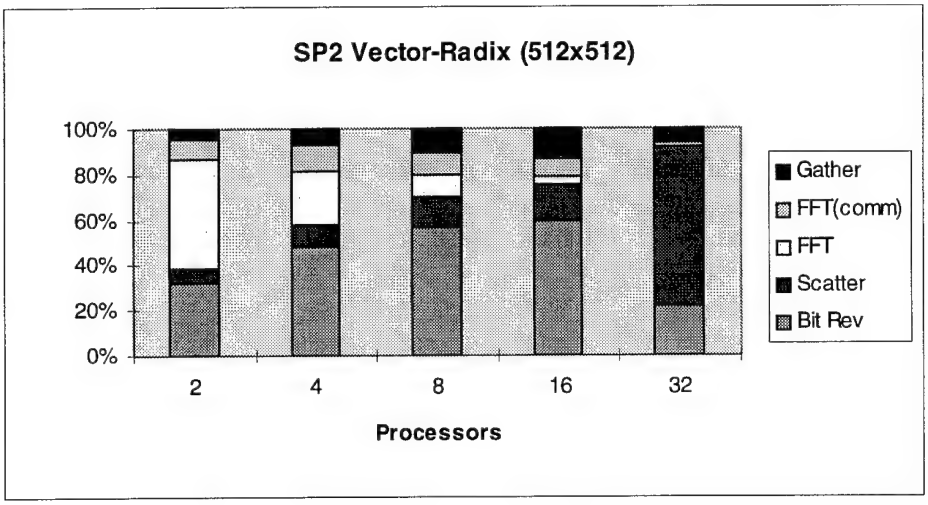


*Figure 4-16. Time Breakdown of Vector-Radix for 512x512 Input Size on the SP2.*

### *4.3.1.1 Optimizations*

The vector-radix optimizations performed well, in general, on the SP2. The first optimization, the packing of rows to avoid the bit-reverse along the rows, was only a minor improvement (less than 10%). The second optimization, LICR and CSE, decreased runtime from 21-44% as the number of processors increased. This is significant because as the problem size increases, the FFT computation becomes a larger percentage of the total runtime, making this improvement's problem size scalability important. The third optimization, avoiding the final permutation with a derived datatype, performed the best of the three optimizations, ranging from 73-91%. Table 4-6

shows the percentage decrease in runtime for the FFT and gather optimizations, as well as the decrease in total runtime. The improvements in overall runtime peak at 16 processors because the scatter becomes the dominant factor in runtime after that point.

*Table 4-6. Percentage Change in Runtime of Optimizations of Vector-Radix on the SP2.*

| Processors | FFT | Gather | Total |
|---|---|---|---|
| **2** | -21.65% | -73.92% | **-54.59%** |
| **4** | -26.56% | -75.67% | **-60.53%** |
| **8** | -27.83% | -80.71% | **-64.74%** |
| **16** | -36.08% | -87.11% | **-88.19%** |
| **32** | -44.15% | -91.15% | **-47.79%** |

#### *4.3.1.2 Scalability*

Figure 4-17 shows the speedup for the different input sizes as the number of processors are increased. The first conclusion that can be drawn from this graph is that very little speedup was obtained, with speedup never reaching 2, regardless of the number of processors. There is an increase in speedup as processors are added, but the increase in far below linear, making the addition of processors only a small improvement. In general, the vector-radix is not a good parallel algorithm on the SP2.

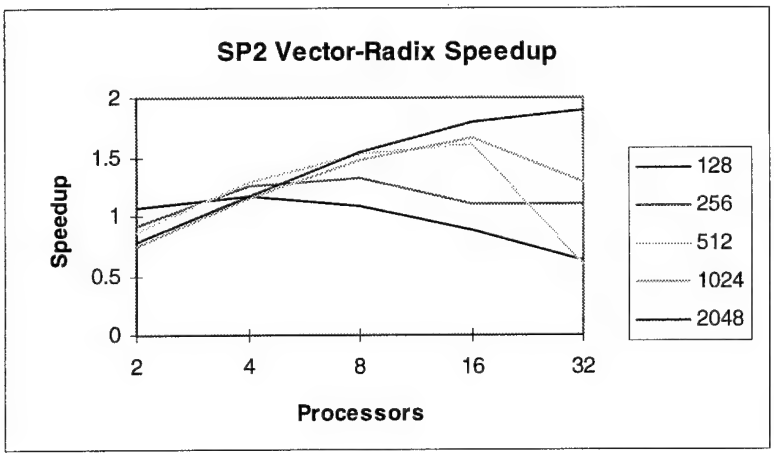


*Figure 4-17. Speedup of Vector-Radix on the SP2.*

### *4.3.2 Row-Column*

The row-column was an improvement over the vector-radix on the SP2, and revealed some surprising optimization results. Figure 4-18 shows the time breakdown of the row-column with four processors. A few conclusions can be made from this figure. First, the final, serial transpose dominates the runtime of the row-column, especially as the problem size increases. As is seen in Section 4.3.2.2, this severely limits the potential speedup of the row-column. Second, the FFT computation comprises approximately 25% of the runtime, regardless of input size. Because the transpose and communication sections dominate the runtime (75%), potential speedup is again limited.

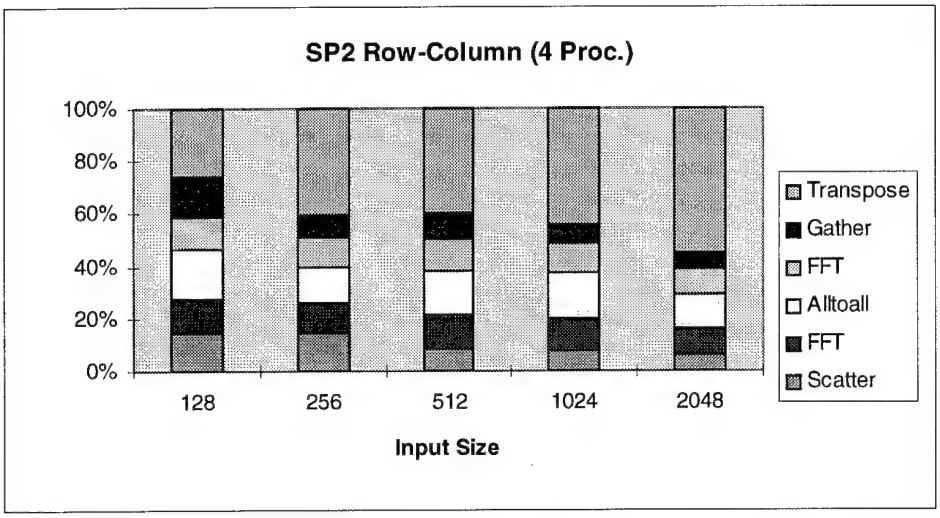


*Figure 4-18. Time Breakdown of Row-Column with 4 Processors on the SP2*

Figure 4-19 shows the time breakdown of a 512x512 input size as the number of processors is increased. The percentage of time spent for the serial transpose again increases as the number of processors increases, pointing to poor scalability. Also, the scatter and gather increase their portion of the runtime as processors increase, while the FFT section shrinks to almost nothing. Again, these trends point to poor scalability because communication and the serial transpose dominate the overall runtime.

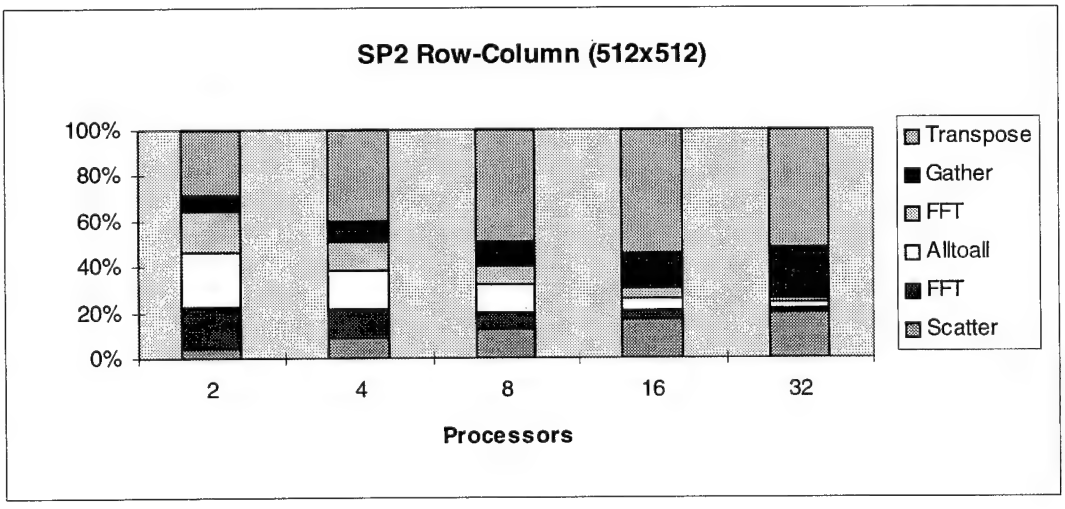


*Figure 4-19. Time Breakdown of Row-Column for 512x512 Input Size on the SP2.*

#### *4.3.2.1 Optimizations*

The row-column optimizations on the SP2 revealed unexpected results. For both optimizations, performance was *lost* instead of gained. First, for the AlltoAll optimization, which replaced the chunk/unchunk process during the matrix transpose step, the unoptimized version was 20-60% *faster* than the "optimized" version with derived datatypes. Second, for the gather optimization, which replaced the gather and final, serial transpose with a gather with a derived datatype, the unoptimized version was anywhere from 30-80% faster. The combination of these two optimizations caused the unoptimized version to run from 15-75% faster overall depending on the input size and number of processors. The performance of the optimizations varied mostly with the number of processors and the peak difference occurred at 8 processors. Table 4-7 shows the average runtime improvement for the two code sections and the overall improvement.

*Table 4-7. Percentage Change in Runtime of Unoptimized Row-Column on the SP2.*

| Processors | All to All | Gather | Total |
|---|---|---|---|
| **2** | -22.16% | -33.48% | **-20.01%** |
| **4** | -46.81% | -52.67% | **-44.03%** |
| **8** | -53.40% | -49.75% | **-43.91%** |
| **16** | -49.61% | -47.63% | **-39.48%** |
| **32** | -49.30% | -41.62% | **-30.91%** |

The conclusion which can be drawn from the row-column optimizations' performance is that derived datatypes which involve transposing of data perform very poorly on the SP2. Since these optimizations performed well on the Paragon and the NOW, there must be either an architectural difference or an implementation difference. It is believed that the SP2's proprietary MPI implementation is the difference because there is nothing unique about the processor to infer this kind of performance. Both the AFIT NOW and the Paragon performed well on these optimizations, and they both use the MPICH framework for their MPI implementation.

#### *4.3.2.2 Scalability*

The row-column on the SP2 shows better speedup than the vector-radix, but speedup is still very limited. Figure 4-20 shows the speedup of the different input size and processors. The speedup only exceeds 2 for one data point and the increase in speedup is far from linear as processors are added. There appears to be a peak in speedup at 8 processors, especially for smaller input sizes. This peak may be explained by the multistage nature of the SP2 network. Processor nodes are divided into groups of 8, called frames, which are interconnected by a single switch. When packets travel between frames, they must traverse a second switch which interconnects the frames. Therefore, when more than 8 processors are used communication times are expected to increase due to switch delays, and hence parallel performance is degraded.

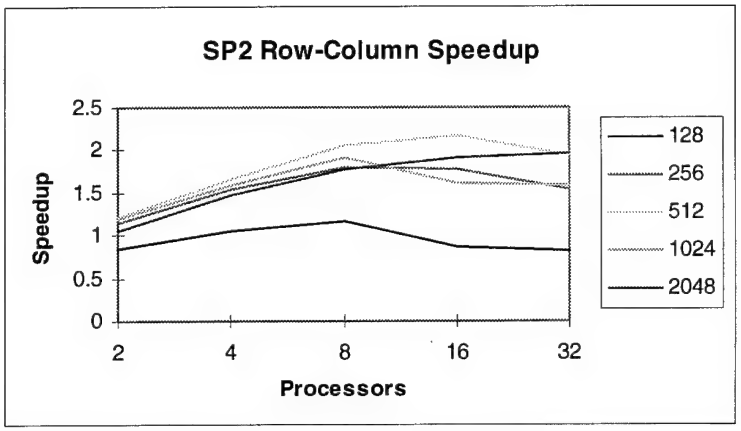


*Figure 4-20. Speedup of Row-Column on the SP2.*

The main problem with the row-column code, especially at large input sizes, is the serial transpose. This transpose is from 40-50% of the total runtime. If the speedup equation is analyzed, the poor performance can be explained. Speedup is defined as,

$$Speedup(S) = \frac{Serial\ Runtime}{Parallel\ Runtime} = \frac{T_s}{T_p} = \frac{T_{FFT} + 2T_{Transpose}}{\frac{T_{FFT}}{p} + T_{Comm} + T_{Transpose}}, \qquad (9)$$

where $T_{FFT}$ is the time for the FFT computation, $T_{Comm}$ is the time for parallel communication, $T_{Transpose}$ is the time for the serial transpose, and p is the number of processors.

Under ideal parallel conditions, let p increase to ∞ and $T_{Comm}$ decrease to zero. In this case, speedup becomes

$$\lim_{p \to \infty} S = 2 + \frac{T_{FFT}}{T_{Transpose}} \qquad (10)$$

If $T_{Transpose}$ dominates $T_{FFT}$, then the speedup is limited to 2, regardless of an increase in the number of processors. Table 4-8 shows the ratio of $T_{FFT}/T_{Transpose}$ on the SP2 for the problem sizes in this research. As the input size increases, this ratio decreases, limiting the potential speedup for the row-column on the SP2. The sharp decrease in the ratio from 128x128 to 256x256 input sizes is explained by cache effects. At 128x128 points, the image is 128 KB, and at 256x256, the image is 512 KB. Since the data cache of the SP2 is 128 KB, it is logical that once the image no longer fits in the cache, that the transpose time would dominate. Because the FFT accesses elements in sequential order, there are fewer cache misses as compared to the transpose which must access elements in column-major order which is not sequential.

*Table 4-8. Ratio of FFT Computation Time to Matrix Transpose Time in Serial Row-Column 2-D FFT on the SP2.*

| Input Size | FFT ($T_{FFT}$) | Transpose ($T_{Transpose}$) | Ratio ($T_{FFT}/T_{Transpose}$) |
|---|---|---|---|
| **128** | 0.005251 | 0.000292 | 17.97279 |
| **256** | 0.022255 | 0.016543 | 1.345299 |
| **512** | 0.104927 | 0.071238 | 1.472921 |
| **1024** | 0.435858 | 0.292235 | 1.491461 |
| **2048** | 1.951272 | 1.762492 | 1.10711 |

### *4.3.3 Pipeline*

The pipeline performed the best of the three algorithms on the SP2. Again, as with the row-column, the optimizations *decreased* the overall performance. This is not

surprising because the pipeline uses similar derived datatypes to avoid serial data transposition. The unoptimized pipeline was, on average, 43% faster than the optimized pipeline with derived datatypes.

The pipeline exhibited good scalability trends as seen in Figure 4-21. The first trend that is visible is good processor scalability up to 9 processors. Until 9 processors, there is a clear improvement when adding processors, but after 9 processors, there is no improvement. The Paragon had the same peak at 9 processors which points to an algorithmic explanation, rather than a platform-specific reason. The peak is best explained by the message traffic of the pipeline as described in Section 3.2.3.3. As the number of processors increases, the number of messages sent and bytes sent increases. There comes a point where the increased traffic dominates the runtime, regardless of the overlapping nature of the pipeline control structure. The second trend seen in the figure is problem size scalability. As the input size increases, speedup is greater or approximately equal, with no peak in speedup for a particular problem size. From these two trends, the pipeline on the SP2 can be considered problem size scaleable and processor scaleable up to 9 processors.

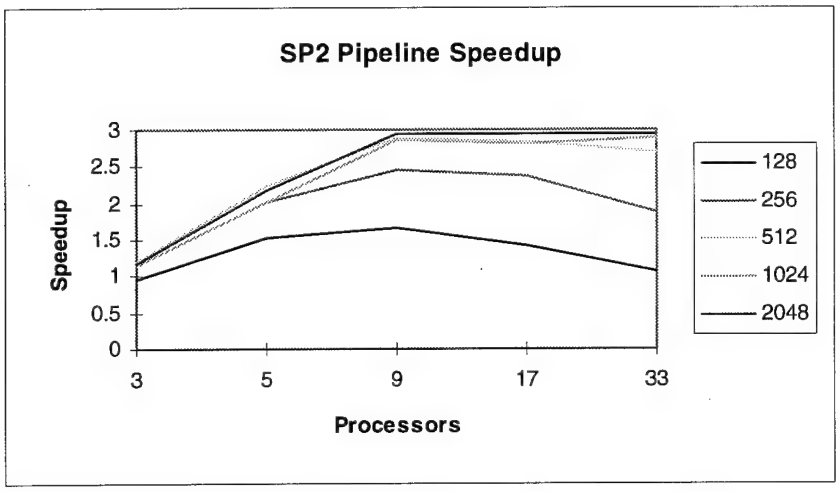


*Figure 4-21. Speedup of the Pipeline on the SP2.*

### *4.3.4 Row-Column vs. Vector-Radix vs. Pipeline*

As with the Paragon, the best algorithm in terms of speedup for the SP2 is the pipeline, as shown in Figure 4-22. The vector-radix and row-column never reached a speedup of 2, which makes them poor candidates for parallelization on the SP2. The pipeline had better speedup, however it did not reach 3 for any problem size or number of processors. The question of whether this is enough speedup for parallelization is qualitative and depends on the user and the application. There are some time-sensitive applications for which any speedup is beneficial, regardless of the number of processors used.

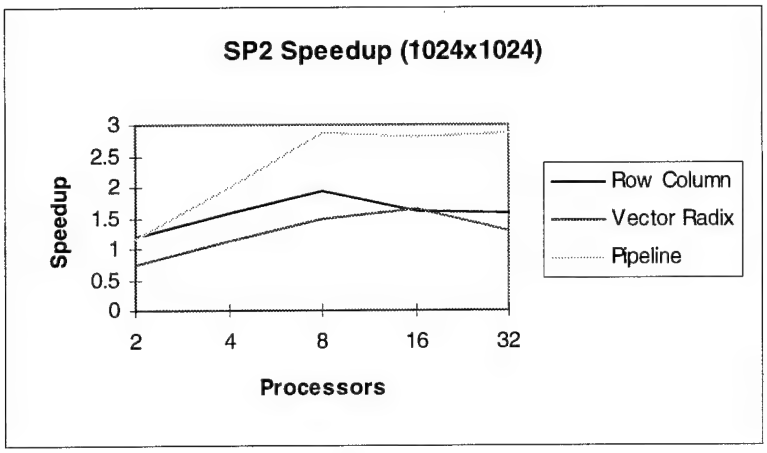


*Figure 4-22. Speedup of the Three Algorithms for 1024x1024 Input Size on the SP2.*

The efficiency graph in Figure 4-23 shows that the pipeline does not match the efficiency of the row-column until 4/5 processors. This occurs because at 2/3 processors, the pipeline uses 50% more processors, but does not have 50% more speedup.

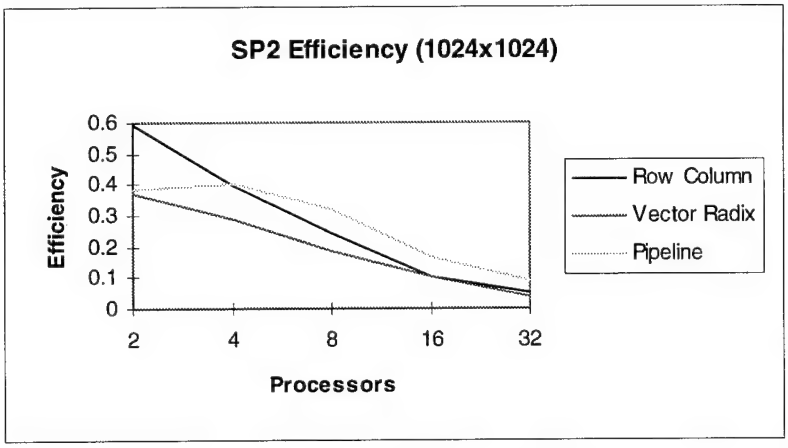


*Figure 4-23. Efficiency of the Three Algorithms for 1024x1024 Input Size on the SP2.*

## *4.4 AFIT NOW*

The purpose of this section is to detail the performance of the AFIT NOW for the three algorithms separately and then compare their performance to determine the best code. It is difficult to draw decisive conclusions about the performance of the AFIT NOW for two reasons. First, because there are only six nodes, the algorithms are limited to 4/5 processors. It is difficult to determine processor scalability as the number of processors increases only from 2/3 to 4/5. Second, because of the problems with the BDM messaging layer as discussed in Section 3.3.5.1, not all the programs worked properly. The incorrect results from collective operations made the unoptimized code unreliable and analysis of the performance of optimizations could not be performed. Also, the pipeline and row-column codes had problems when using 4/5 processors. The row-column locked up for input sizes greater than 1024x1024 and the pipeline locked-up at 2048x2048. Because of these two problems, there are only 8 "valid" data points where all three algorithms worked correctly on the AFIT NOW.

### *4.4.1 Effect of Network and Messaging Layers on Performance*

From the discussion in Section 2.2.3, the AFIT NOW was not expected to perform well using Ethernet as the interconnection network. After only a few tests, it was obvious that the slow network (1.2 MB/sec) was the bottleneck. Using TCP/IP as the messaging layer for Ethernet contributes to this problem. It was expected that using a high bandwidth, low latency network such as Myrinet would greatly improve performance over Ethernet. This improvement occurred, but there is still a bottleneck at the messaging

layer, as discussed in Section 2.2.4. Table 4-9 shows the performance of Ethernet with TCP/IP, Myrinet with TCP/IP, and Myrinet with the BDM messaging layer on a 256x256 2-D FFT with 4/5 processors. From this table, it is obvious that only the combination of a fast network *and* a low overhead messaging layer can allow NOWs to reach their full potential.

*Table 4-9. Performance in Megaflops of the Three Algorithms with Different Network and Messaging Layers on the AFIT NOW.*

| Algorithm | Ethernet (TCP/IP) | Myrinet (TCP/IP) | **Myrinet (BDM)** |
|---|---|---|---|
| Vector Radix | **2.2** | **35.5** | 76.1 |
| Row Column | **2.2** | **43.8** | 95.6 |
| **Pipeline** | **2.4** | **51.3** | 151.9 |

### *4.4.2 Vector-Radix*

The vector-radix performed the worst of the three algorithms by far on the AFIT NOW. Figure 4-24 shows the breakdown of the vector-radix for 2 and 4 processors. The 2 processor case is consistent regardless of input size, with no significant trends in the data. At the 4 processor case, however, an important trend is seen. At input sizes of 512x512 and greater, the FFT communication, during which processors swap half of their slice of the image, becomes a greater percentage of runtime. This anomaly is not supported by any platform explanation, it is believed to be the result of an immature

MPI/messaging layer implementation. This problem is further discussed in Section 4.4.3 because the row-column code had similar problems.

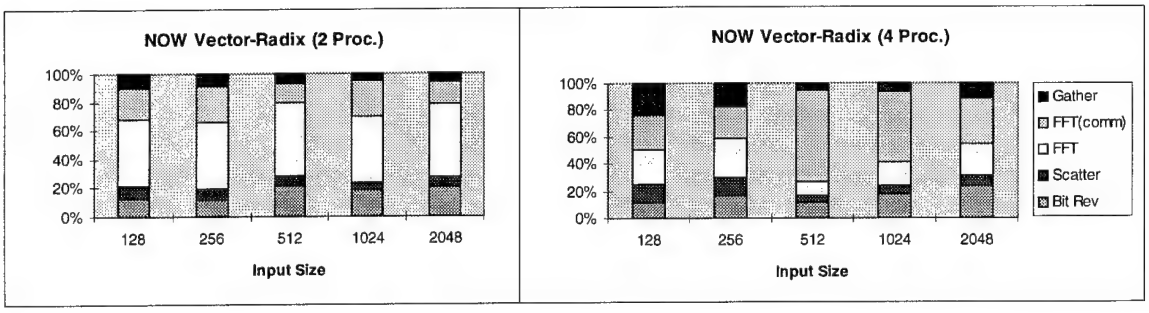


*Figure 4-24. Time Breakdown of Vector-Radix with 2 and 4 Processors on the AFIT NOW.*

The vector-radix shows no speedup over a serial implementation as seen in Figure 4-25, making it a poor choice for parallelization. As the input size increases, the speedup decreases, so even an increase in problem size does not increase speedup as might be expected. There are two interesting trends to note which are also seen in the row-column implementation. First, the 128x128 problem size shows the highest speedup of the different problems sizes. Second, at the 512x512 problem size with 4 processors, there is a sharp drop in speedup.

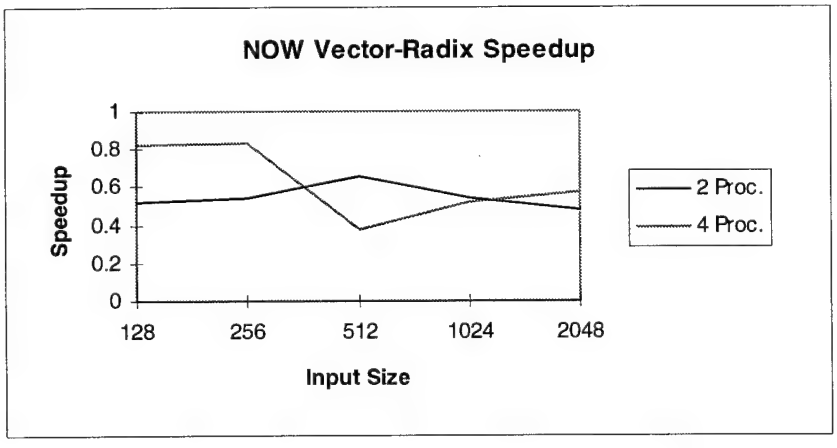


*Figure 4-25. Speedup of Vector-Radix on the AFIT NOW.*

### *4.4.3 Row-Column*

The row-column also exhibited poor performance on the AFIT NOW. Figure 4-26 shows the time breakdown of the row-column with 2 and 4 processors. As with the vector-radix, the 2 processor case is consistent across all input sizes. With 4 processors, however, the AlltoAll becomes dominant at 512x512. The row-column AlltoAll code and the vector-radix FFT computation code both increase their percentage of total runtime at the 512x512 input size. The best explanation of this problem is that both of these sections of code use the MPI_Sendrecv function, which is used to exchange data between processors. In fact, it is the MPI_Sendrecv which locks-up in the row-column with 4 processors and input sizes above 512x512. This problem seems to point to a incorrectly written MPI implementation or messaging layer. This is not completely unexpected since this is just the first release of BDM and it is part of an ongoing graduate research project. In fact, in their supporting documentation, they state that the code has not been thoroughly tested for correctness.

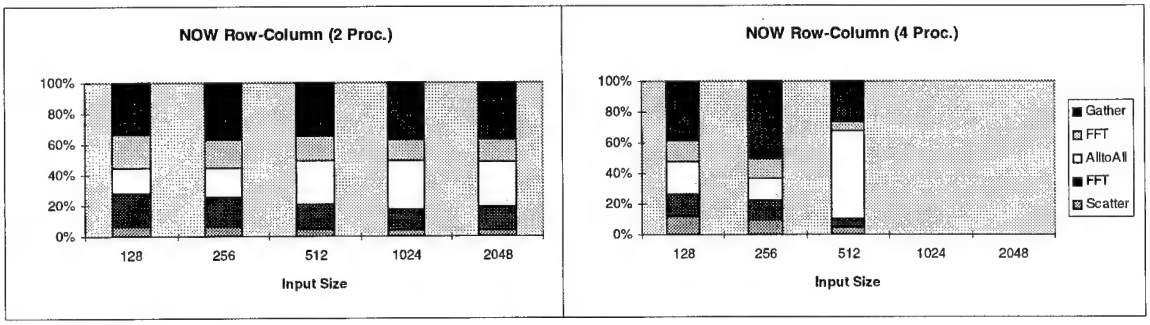


*Figure 4-26. Time Breakdown of Row-Column with 2 and 4 Processors on the AFIT NOW.*

The row-column, in a similar fashion to the vector-radix, showed poor speedup as seen in Figure 4-27. Unlike the vector-radix, some speedup (1.15) was obtained with 4 processors on the 128x128 problem size. Again, the speedup peaks at the 128x128 problem size and there is a sharp drop with 4 processors at 512x512 points.

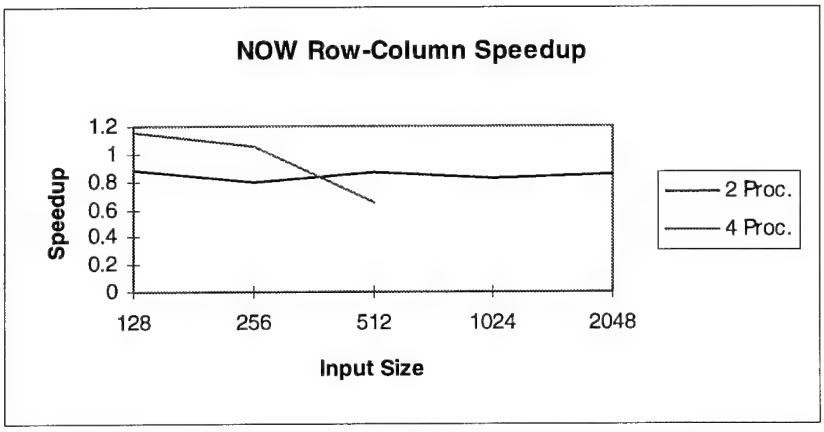


*Figure 4-27. Speedup of Row-Column on the AFIT NOW.*

### *4.4.4 Pipeline*

As disappointing as the vector-radix and row-column performed, the pipeline performed very well on the AFIT NOW. The pipeline exhibited the best processor scalability of the three algorithms, increasing performance with additional processors as seen in Figure 4-28. As with the row-column, though, the peak performance occurred at 128x128 points, so the pipe did not show problem size scalability. Unfortunately, the pipe also had a problem with locking-up at the 2048x2048 input size with 5 processors.

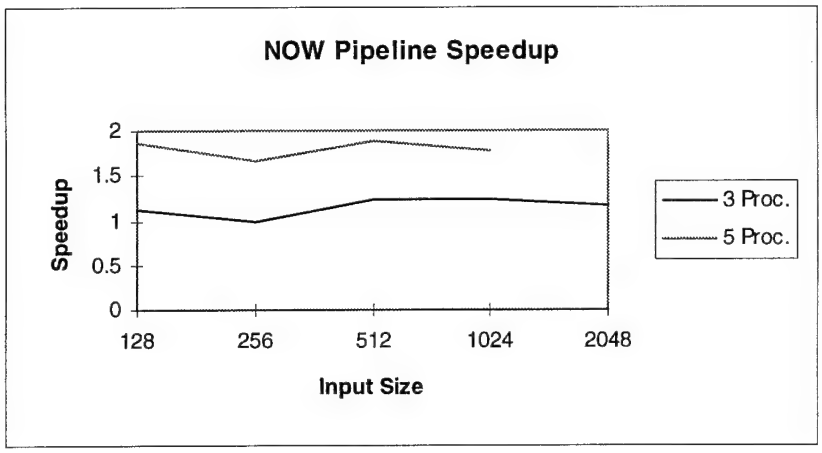


*Figure 4-28. Speedup of Pipeline on the AFIT NOW.*

The good performance of the pipeline on the AFIT NOW is best explained by the time breakdown of computation and communication. Figure 4-29 shows that the computation is approximately 40% of the total runtime with 3 processors. With 5 processors, the FFT begins to decrease its percentage of runtime from 40% to 25% which explains the drop in performance as input size increases.

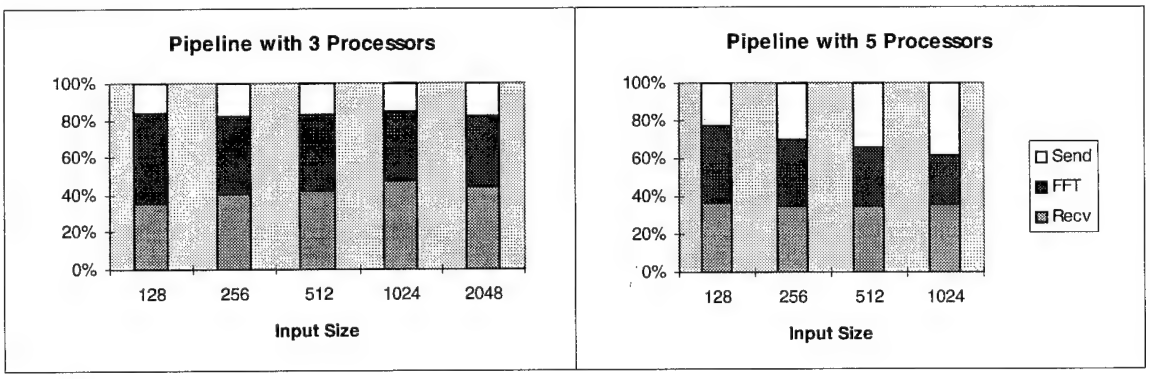


*Figure 4-29. Time Breakdown of Pipeline with 3 and 5 Processors on the AFIT NOW.*

### *4.4.5 Vector-Radix vs. Row-Column vs. Pipeline*

When comparing the three algorithms, some conclusions can be made despite the dearth of data. The vector-radix did not perform well for any problem size or number of processors. It exhibited no speedup and was at least 80% slower than the best code. When comparing speedup, the pipeline always performed the best, by at least 22% over the row-column code. Figure 4-30 shows the speedup curves of the three algorithms for 2 and 4 processors. Only the pipeline consistently exhibited speedup greater than one, making it the best choice for parallelization on the AFIT NOW.

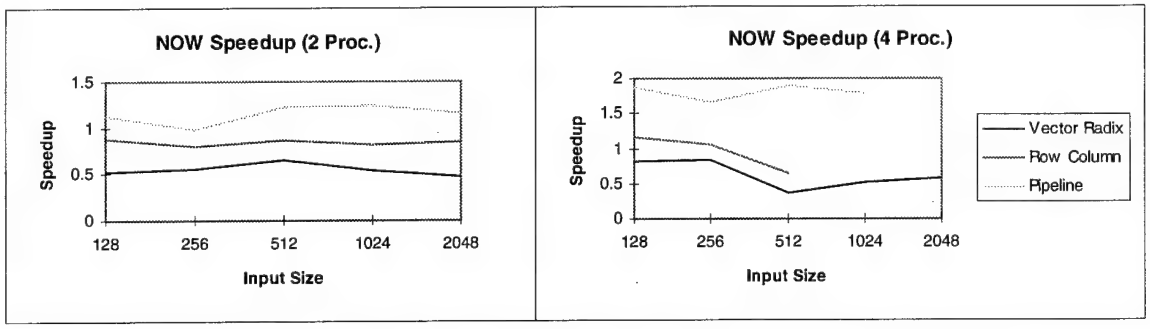


*Figure 4-30. Speedup of the Three Algorithms with 2 and 4 Processors on the AFIT NOW.*

When comparing the three algorithms in terms of efficiency, which takes the number of processors used into account, the row-column was competitive with the pipeline at the 2/3 processor case as seen in Figure 4-31. This occurs, as it did with the Paragon, because of the relationship between the number or processors and efficiency. Since the pipeline uses 50% more processors than the row-column, it would have to have 50% greater speedup to have equal efficiency. As the number of processors increased to

4/5, the pipeline again regained its dominance as the best algorithm for the AFIT NOW, similar to the Paragon.

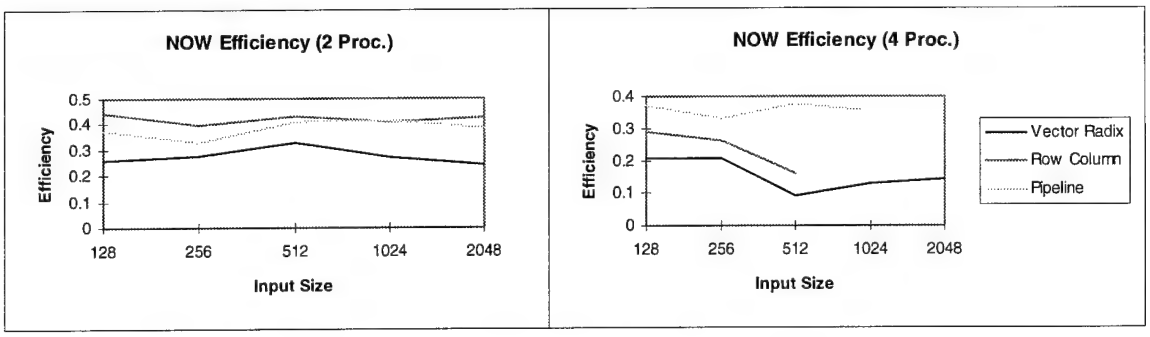


*Figure 4-31. Efficiency of the Three Algorithms with 2 and 4 Processors on the AFIT NOW.*

In summary, there are a few conclusions which can be drawn about the AFIT NOW. First, BDM is an experimental messaging layer and therefore is subject to problems with both effectiveness and efficiency. Particularly, the send/receive functions appear to have performance and deadlock problems, and the collective operations have correctness problems. It is hoped that these problems will be fixed in the near future, or another more viable messaging layer will become available (possible Berkeley's AM).

Second, BDM performed best at small message sizes because performance decreased as the input size increases. This trend is directly related to the design choices discussed in Section 2.2.4. When transferring a message from main memory to the Myrinet NIC (Lanai), only programmed I/O (loads and stores) are used. This choice was made based on the assumption that fast processors can perform programmed I/O at a speed near that of DMA transfers. Also, programmed I/O avoids the overhead of DMA setup and extra copying of data, and hence improves latency, possibly at the cost of lower

throughput. This design choice is in contrast to FM's choice of programmed I/O from the host to the Lanai and DMA from the Lanai to the host. The result of these different design choices is different problem size scalability. For all three algorithms, the AFIT NOW with BDM had its best performance at the 128x128 input size. For the AFIT NOW with FM* , the opposite trend is seen. for FM, as the input size increases, the row-column with 2 processors improves from 52 to 59 megaflops, and the row-column with 4 processors improves from 24 to 82 megaflops. It is evident from these performance results that the messaging layer's design choices and the tradeoff between low latency and high throughput strongly effect performance.

### *4.5 Paragon vs. SP2 vs. AFIT NOW*

The purpose of this section is to compare the three platforms for each algorithm individually. This comparison is useful if one particular algorithm is going to be used exclusively. For example, if the pipeline was chosen as the single 2-D FFT algorithm, a comparison of the platforms for this particular algorithm could be used to select a platform.

---

* Despite FM's problems with derived datatypes, some results were gathered. While these performance results cannot be validated given the correctness problems, it can give a general estimate of the performance of FM.

### *4.5.1 Vector-Radix*

The vector-radix, which generally performed the worst of the three algorithms on all the platforms, gave a clear ordering of the platforms in terms of runtime. In both the 2 and 4 processor cases, the SP2 had the best runtime, followed by the AFIT NOW and the Paragon. For all data sizes, the NOW was 40-60% slower, except at 256x256 input size where it was within 20% of the SP2. The vector-radix results were generally unimpressive because the Paragon was significantly slower, and the SP2 and AFIT NOW showed almost no speedup. As a result, the vector-radix does not appear to be an interesting algorithm for high performance signal processing, given today's computational environment.

### *4.5.2 Row-Column*

The row-column algorithm proved to be a better algorithm for the 2-D FFT than the vector-radix, but lacked the performance of the pipeline. Figure 4-32 shows the performance (in megaflops) of the row-column for each platform, with 2 and 4 processors. The SP2 exhibits the highest megaflops of the three platforms, but the AFIT NOW is competitive (within 10%) at small input sizes. The AFIT NOW's performance at small input sizes corresponds with the design of BDM for low latency. A different messaging layer would probably exhibit different trends and possible would equal the performance of the SP2 for large input sizes. For example, with FM, the AFIT NOW

reaches 82 megaflops at 2048x2048 with 4 processors, which is within 10% of the SP2's performance.

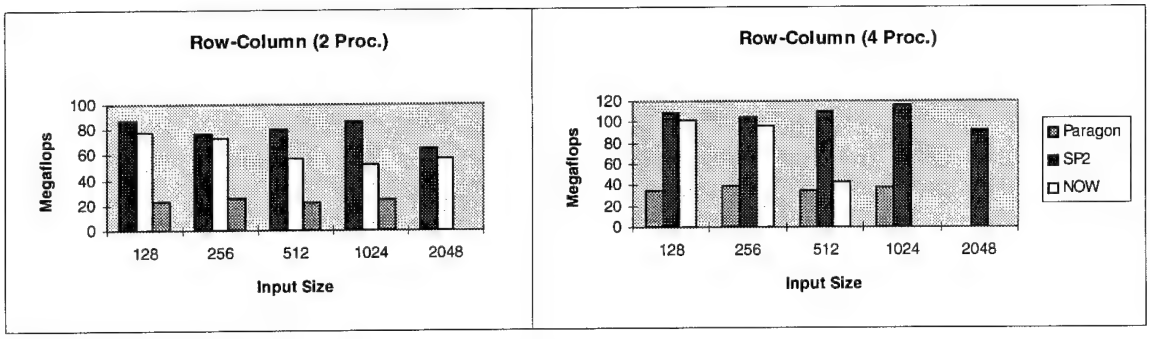


*Figure 4-32. Performance in Megaflops of Row-Column with 2 and 4 Processors on the Three Platforms.*

### *4.5.3 Pipeline*

The pipeline algorithm was the best algorithm and was also where the AFIT NOW made performance gains on the SP2. Figure 4-33 shows the performance (in megaflops) of the pipeline on the three platforms. The Paragon again had the worst performance, with only a fraction of the megaflops of the other two platforms. The AFIT NOW and SP2 had near equal performance with only a 1% average difference in runtime between the two platforms.

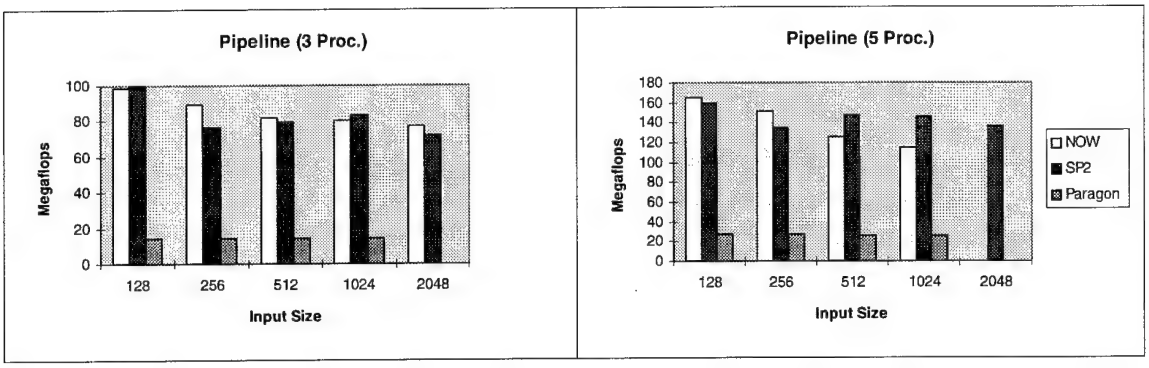


*Figure 4-33. Performance in Megaflops of the Pipeline with 3 and 5 Processors on the Three Platforms.*

## *4.6 Best Algorithm*

This section discusses the best 2-D FFT algorithm across platforms, input sizes and numbers of processors. When this research was begun, it was expected that the best algorithm would vary depending on the various factors discussed. In reality, the pipeline exhibited the best performance except for a small number of cases. Table 4-10 shows the average decrease in runtime for the pipeline compared to the vector-radix and row-column across all input sizes and numbers of processors.

*Table 4-10. Average Decrease in Runtime of Pipeline vs. Row-Column and Vector-Radix for the Three Platforms.*

| *Platform* | *Row-Column* | *Vector-Radix* |
|---|---|---|
| Paragon | 34% | 53% |
| SP2 | 25% | 43% |
| AFIT NOW | 34% | 56% |
| **Average** | 31% | 50% |

It is interesting to observe that the pipeline performed the best on all the platforms despite them having completely opposite computational and communication performance. The Paragon has approximately one-fourth the computational power of the AFIT NOW, but has over two and one-half times the network throughput. The pipeline has two characteristics which benefit these two very different platforms. On the Paragon, the pipeline can hide the FFT computation latency by overlapping communication and computation. On the AFIT NOW, the computation to communication ratio is kept high because the processors operate on larger data sizes than in the row-column or vector-radix. By keeping this ratio high, the effects of relatively slow communication is minimized.

### *4.7 Best Platform*

To make the most fair comparison of the three platforms, the best times for each input size and number of processors was used. Using a single algorithm could tilt the comparison in favor of one platform or another because the algorithms perform very differently on different platforms. For example, if the vector-radix algorithm was used, the SP2 would outperform the AFIT NOW in all cases. If the pipeline was used, the two platforms would be considered equal. To avoid this problem, the best times from each platform, regardless of the algorithm, are used to evaluate the comparative performance of the three platforms.

Runtime is the best measure of performance when comparing platforms because it measures the time seen by the user when running an algorithm. When analyzing the runtime of the best algorithm on each platform, two trends are evident as seen in Figure 4-34. First, the Paragon was approximately 60% slower than the SP2 or AFIT NOW. Second, the SP2 and AFIT NOW had almost the same performance. In fact, the average runtime difference between the two is 1.5% excluding the points where the AFIT NOW had messaging layer problems.

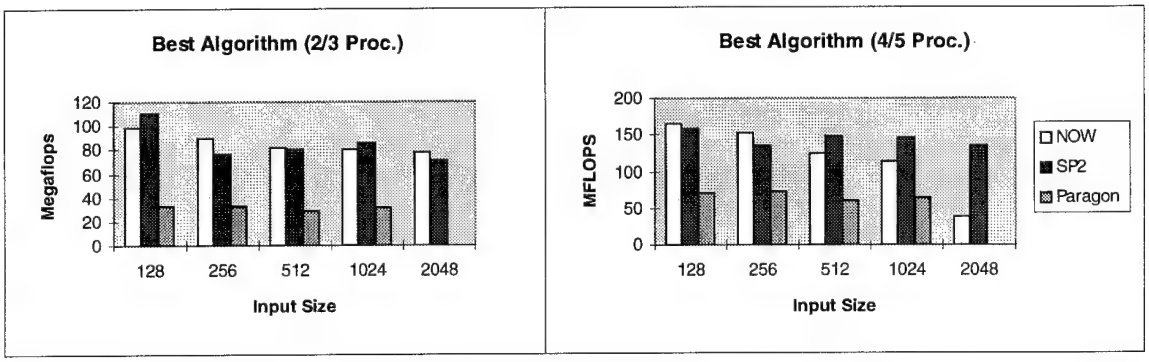


*Figure 4-34. Performance in Megaflops of the Best Algorithm on Each Platform with 2/3 and 4/5 Processors.*

To better see the gap between the performance of the Paragon and the AFIT NOW, Table 4-11 shows the best Paragon parallel runtimes and the best AFIT NOW serial runtimes for varying problem size. The result of this negligible difference is that the Paragon, despite having superior speedup and efficiency, did not perform much better than a single, state-of-the-art workstation for the 2-D FFT. Despite having much better communication, the Paragon could not overcome its poor computation performance, due to its older processor. This difference solidifies the choice of workstations as an alternative platform to the Paragon for this application.

*Table 4-11. Runtime of Paragon Pipeline with 9 processors and Serial 2-D FFT on 200 MHz Ultra.*

| Input Size | Paragon (sec) | Ultra 200 (sec) | Percentage Difference |
|---|---|---|---|
| 128 | **0.011** | **0.012** | **-10.28%** |
| 256 | **0.057** | **0.053** | **7.91%** |
| 512 | **0.345** | **0.329** | **4.81%** |
| 1024 | **1.364** | **1.473** | **-7.42%** |

# *V. Conclusion*

Multiple processor systems have become inevitable as the size of problems which need to be solved has increased. Some computer architects have proposed networks of workstations, using COTS technology, as a replacement for more expensive massively parallel processors. This research has attempted to quantify the differences between these approaches by comparing their performance for the 2-D FFT signal processing application. This chapter discusses the viability of parallelization of the 2-D FFT, the performance of the various algorithms, and a summary of NOW and MPP performance. The contributions of this research are also presented, along with recommendations for future work.

## *5.1 Conclusions*

Chapter IV detailed a wealth of trends and anomalies in the performance data collected from a variety of algorithms on different platforms. This section attempts to take a higher level view of the results and draw more general conclusions about the 2-D FFT application and the computational platforms on which it is used.

### *5.1.1 Parallel 2-D FFT Performance*

The data presented in Chapter IV leads to important conclusions about the value of parallelization of the 2-D FFT. For an older platform, such as the Paragon, with its high communication-to-computation ratio, parallelization is always beneficial because speedup is displayed, and usually increases as processors are added. However, for today's

computational platforms such as the SP2 and NOWs, the communication-to-computation ratio is considerably lower and very little speedup is seen, regardless of the number of processors. This poor scalability undermines the use of large number of processors and hence the use of *Massively* Parallel Processors. Beyond that, however, the poor speedup brings into question whether parallelization is beneficial at all for the 2-D FFT or related signal processing algorithms. If speedup is limited, even with a large number of processors, then parallelization would seem inefficient and possibly unnecessary.

Isoefficiency analysis purports that speedup can be obtained by increasing the problem size. The results showed, however, that quite often speedup peaks at a problem size less than 2048x2048. As a result, this type of asymptotic analysis may not provide a realistic model of computation. These types of conclusions made about parallel processing based on machines such as the Paragon may not hold true for the changing face of computational platforms. If processor performance continues to improve without bound, the time spent in computation in programs will decrease. If communication performance does not improve at an equal rate, then communication overhead will increase and as a result, speedup will drop. Algorithms which once obtained speedup, may no longer be considered parallelizable on modern computational platforms. Only those problems with fewer communication requirements will continue to obtain speedup, given this changing ratio of communication to computation.

The results of Chapter IV also question the use of the speedup metric as an indicator of parallel performance. Speedup is indispensable when determining the

scalability of an algorithm or when deciding whether parallelization is effective. Speedup is misleading, however, when used in lieu of runtime for comparing platforms. This problem was evident from the platform analysis. The Paragon always exhibited speedup (sometimes even superlinear) for all the algorithms, regardless of problem size of the number of processors. The SP2 and NOW exhibited very limited speedup and no speedup at all for some cases. The maximum speedup realized by the Paragon was 7.8, while the SP2 reached only 3.0, and the AFIT NOW managed only 1.9. If speedup was reported, the Paragon would appear to be the best computational platform with many times the speedup of the other two platforms. This data is meaningless to the end-user, however, because the Paragon, with up to 32 processors, cannot even outperform a single Ultra Sparc workstation. It is obvious that speedup is not a true representation of platform performance, but clouds the true differences between platforms.

### *5.1.2 2-D FFT Algorithms*

The previous section strongly questioned the value of parallelizing the 2-D FFT on modern processors. Despite this conclusion, there is still interest in choosing the best parallel algorithm. There are users who have applications such as real-time signal processing, which need to use parallel platforms in an attempt to achieve any possible speedup. Also, there may be unforeseen technological changes in the future which would change the communication-to-computation ratio in favor of parallelization . For these reasons, the analysis of parallel 2-D FFT algorithms is presented.

The three algorithms that were implemented, the vector-radix, row-column, and pipeline, have different attributes which make them the best choice depending on the platform, number of processors employed, and the size of the image. The number of computations, the memory access patterns, and the communication patterns all have varying effects on the performance of an application. Given these differences, the signal processing system designer has two choices. First, a hybrid could be developed, which chooses the best implementation in an automated fashion, based on either performance measurement or heuristics. A technique similar to the FFTW approach could be used, which would test different algorithms for the given image size and number of processors to find the optimal implementation on a given platform. Unfortunately, for large image sizes, this may prove time-consuming which would limit its suitability for production use. An approach similar to the Cornell approach could be used to build models of performance based on a small set of empirical data.[Durie97] From these models, the best implementation could be chosen at runtime for the given platform, problem size and number of processors.

A second option for choosing parallel algorithms is to use research such as this thesis effort to find a near optimal algorithm for a majority of image sizes and numbers or processors. This approach is somewhat perilous because this research is limited to certain input sizes and processors. From the data collected, however, it appears that the pipeline may be the best such algorithm for today's computation platforms. On platforms which have high computational performance with relatively low communication performance (NOWs and SP2), the communication patterns and larger problem sizes of the pipeline

have advantages over those of the vector-radix and row-column. For platforms with high communication performance and low computation performance (Paragon), the FFT computation time can be hidden by the overlapping of computation and communication. Given these results, the pipeline may prove to be the best algorithm for both MPPs and NOWs for real-time signal processing.

The comparison of the three algorithms yielded different results than reported in some research papers. In the research by An et al.[An95] they conclude that the vector-radix implementation outperforms the row-column on the Paragon. Two major differences between An's framework and this research are the algorithm implementations and the use of hand-tuned assembly code. While their implementations are not detailed, their vector-radix algorithm does not perform the initial bit-reverse along the rows and columns which was from 10-50% of the total runtime depending on the input size and number of processors. When the 2-D FFT is part of a signal processing system, the bit-reverse cost can be amortized across many stages. However, in this research, in order to have a correct, stand-alone 2-D FFT the bit-reverse was necessary. Their performance results are also significantly faster (approximately 50%) than those reported in this research. This is not surprising, however, since assembly code routines are used for all computations and the communication library used is more than likely Nx as opposed to the portable, less efficient MPI. In Patel and Jamieson's research,[Patel93] the vector-radix is shown to be superior for small problem sizes (approximately 8-12 rows per processor) on the Intel Touchstone Delta. These problem sizes are smaller than the limits of this research, so the results concur with the row-column being superior to the vector-radix on

the Paragon. In addition, the results of this research also show that the vector-radix is more competitive with the row-column as the per node problem size decreases, which agrees with Patel's findings. The runtimes reported by this thesis effort are within a range of plus or minus 30% of Patel's results for most input sizes and processors, despite the fact that their row-column does not perform the final matrix transposition which may account for up to 35% of the total runtime. In summary, this research supports the conclusions of Patel and Jamieson, and is different than those of An et al., which is explained by differences in the comparative framework and algorithm implementations.

In order to provide the sponsor with the best parallel 2-D FFT code, numerous optimizations were considered and implemented. It was found that these optimizations can significantly improve performance, but seem to be dependent on the MPI implementation used. On the platforms which used MPICH (Paragon and NOW), the optimizations almost always improved performance. On the SP2, which used a proprietary MPI implementation, the optimizations, especially those with derived datatypes, performed poorly. Regardless of these problems, a thorough understanding of the optimizations available is paramount to providing the highest performance parallel signal processing code.

In summary, to determine the best 2-D FFT implementation, it is important to take into account the algorithm used, the image size, the number of processors, and all aspects of the platform, including the communication and computation speed, the communication

libraries, and the memory hierarchy. Only by understanding the entirety of these performance factors can the best 2-D FFT be chosen.

### *5.1.3 Computational Platforms*

The performance data of the different algorithms on the SP2, Paragon, and AFIT NOW provides a comparison tool for evaluating these platforms. Despite showing outstanding (sometimes superlinear) speedup, the Paragon does not appear to be a competitive platform for this application. Even when using as many as 32 processors, the Paragon had roughly the same runtime performance as a *single* Sparc Ultra workstation. Despite its superior communication infrastructure, the Paragon cannot overcome the poor performance of its older processor.

When choosing a platform to use for signal processing, the SP2 and AFIT NOW offer similar serial and parallel performance. On serial 2-D FFTs, they perform nearly the same, despite the SP2's higher performance on the SPEC benchmark. The computational parity between the SP2 and AFIT NOW is surprising because NOWs are expected to have faster processors. This difference can be attributed to good system design by IBM in using the RS/6000 line of processors.

When comparing the parallel program performance of the SP2 and AFIT NOW, the results must be tempered by three factors. First, comparisons were made only with up to four processors (five with pipeline) because of the limitations of the AFIT NOW. This is not as big a problem as it first may seem, because on the SP2 and Paragon,

performance peaked at eight processors in most cases, with only a small increase in performance between four and eight. Since it appears the 2-D FFT is not processor scaleable beyond eight processors, comparing up to the four processor case represents a majority of the interesting problem space. Second, the poor performance of optimizations using derived datatypes on the SP2 hurt its performance. Given a different MPI implementation, the SP2 is expected to perform better. Third, the problems with the messaging layers on the AFIT NOW, especially for larger data sizes, hurt its performance. With a more reliable messaging layer, it is expected the AFIT NOW would perform better for a wider range of problem sizes.

Given these three limitations, the analysis does provide an approximation of the ability of NOWs to compete with MPPs for signal processing. In all cases, the AFIT NOW was competitive with the SP2 in terms of runtime performance. Ignoring the anomalous data points from messaging layer problems, the NOW ranged from 15% slower to 16% *faster* than the SP2. These results point to, at the least, near equal performance for NOWs, and at the most, a better computational platform.

The results achieved from this research can begin to draw conclusions about the performance of MPPs and NOWs. A more significant figure of merit for those investing in computational platforms would be a cost/performance comparison to determine the best "bang for the buck." From this study, NOWs appear to offer, at the very least, 85% of the performance of MPPs for this application. Yet NOWs cost can be estimated at 50% of the cost of MPPs for a large system (128 nodes).[Anderson95] These figures do not

even take into account the other advantages of NOWs such as heterogeneity, maintenance costs and availability for interactive use. This superior cost/performance ratio makes NOWs a more attractive platform for parallel programming.

## *5.2 Contributions*

This research has made contributions in both the signal processing and computer architecture domains, as well as provided the sponsor with high performance 2-D FFT code. These contributions include the following:

- Three parallel 2-D FFT implementations were developed from existing code using the portable MPI communication library. These three implementations provide the sponsor with alternatives for choosing a 2-D FFT given their platform and programming environment. An important conclusion derived from these findings is that the pipeline implementation, which has been sparsely discussed in the research on signal processing, may be the most advantageous algorithm for today's computational platforms.

- Numerous optimizations were investigated for the three implementations, which led to high performance code. Derived datatypes, non-blocking communication, and compiler-like optimization techniques decreased runtime as much as 90%. Through the understanding and proper use of these optimizations, programmers using a similar framework can greatly improve the performance of their parallel signal processing code.

- The data collected on the three implementations provided the basis for an application-driven comparison of NOWs and MPPs. Much research has been focused on the separate computation and communication performance of these two types of platforms. Only when computation and communication are combined in an application domain, can the difference seen by the end-user be uncovered. This research melds the two areas by using the parallel 2-D FFT to provide a more realistic comparison of MPPs and state-of-the-art NOWs.

- An evaluation of the performance of messaging layers on NOWs has been presented. These results quantify the differences between TCP/IP and more efficient messaging layers. Also, the performance differences between two messaging layers have been related to the design choices made in their development.

- An analysis of the changing computational environment and its effect on computational problems has been presented. A smaller communication-to-computation ratio for today's platforms has reduced the class of algorithms which are parallelizable. Given these changes, it appears that the 2-D FFT may no longer realize great benefits from parallelization because of the limited speedup observed.

### *5.3 Recommendations for Future Work*

There are two mains areas of research which have the potential for future work. In the signal processing domain, the pipeline algorithm has qualities which are beneficial

for many types of parallel platforms. Further study of the pipeline for not only the 2-D FFT, but a wider range of applications may lead to improved parallel performance in many areas. Also, a study of more signal processing algorithms could be performed to determine if the conclusions drawn about the performance of the 2-D FFT extend other algorithms. Finally, a multi-stage signal processing system, which includes other applications such as QR factorization or LU decomposition could be developed to study the effects of keeping data distributed between stages, rather than redistributing the data before and after computation as was done with the 2-D FFT.

In the computer architecture domain, the new messaging layers for NOWs are still in their infancy, as demonstrated by the problems encountered with them on the AFIT NOW. Assuming their development continues, the comparison between MPPs and NOWs could be made clearer with more reliable performance results. Also, a larger NOW would provide better insight into the scalability of the Myrinet network, especially if multiple switches are interconnected. Finally, a study of other application domains other than signal processing would provide data to better evaluate the changing computational environment.

### *5.4 Future of Networks of Workstations*

The viability of NOWs is evident, but there are still obstacles that must be overcome in order for NOWs to move from a research topic to a production environment. First, efficient messaging layers, which are the key to narrowing the gap between

hardware and measured communication performance, must continue to be developed. The few research projects which have produced the messaging software must continue to mature and overcome their correctness and performance problems. Only when reliable and efficient messaging layers are available will NOWs consistently surpass MPPs in performance and reliability. The second challenge for NOWs is a global operating system to present users with a black box view of the platform on which to run their parallel or serial programs. Berkeley's AM project should offer the first prototype of such an operating system in early 1998. The third challenge for NOWs is to overcome the I/O bus bottleneck which hinders its communication performance. While this problem was seen on the Sun platforms (sBus), it is acknowledged that most workstations have a similar deficiency. New I/O bus standards such as PCI have the potential to increase communication performance by at least 50%.[Myricom97] While both MPPs and NOWs will benefit from this change, NOWs should always realize these improvements sooner with quicker time to desktop than the more complex MPP systems.

Given the current state of MPPs and NOWs, what does the future hold for these two approaches? In reality, the NOW/MPP dichotomy is dissolving as the advantages of both approaches are being combined. MPP manufacturers are using more COTS technology in an attempt to lower cost and decrease time to market. NOW researchers are developing more complex operating system software to bundle the NOW platform in an MPP-like package, which adds to the cost and design time of NOWs. More and more, MPPs will look like NOWs and vice versa, leading to a unified approach to parallel platform development based on the balance of cost and performance.

## Bibliography

[An95] M. An et al., "Comparison of 2-D FFT implementations on the Intel Paragon massively parallel supercomputer," *Proceedings of the 1995 International Conference on Acoustics, Speech, and Signal Processing*, vol. 4, pp. 2755-8.

[Anderson95] T. Anderson, D. Culler and D. Patterson, "A case for NOW," *IEEE Micro*, vol. 15, no. 1, pp. 54-64, Feb. 1995.

[Boden95] N. Boden, et al., "Myrinet - a gigabit per second local area network," *IEEE Micro*, vol. 15, no. 1, pp. 29-36, Feb. 1995.

[Bracewell86] R. Bracewell, *The Harley Transform.* New York: Oxford Press, 1986.

[Chan92] S. Chan and K. Ho, "Split vector-radix fast Fourier transform," *IEEE Transactions on Signal Processing*, vol. 40, no. 8, pp. 2029-2039, August 1992.

[CHIMP97] CHIMP home page - http://www.epcc.ed.ac.uk/pg/CHIMP.

[CHSSI97] CHSSI Program description available at http://hpcmo.hpc.mil/Htdocs/CHSSI

[Cooley65] J. Cooley and J. Tukey, "An algorithm for the machine calculation of complex Fourier series," *Mathematical Computation*, vol. 19, pp. 297-301, 1965.

[Culler96] D. Culler et al., "Assessing fast network interfaces," IEEE Micro, vol. 16, no. 1, Feb 1996.

[Duhamel84] P. Duhamel and H. Hollmann, "Split radix FFT algorithm," *Electronic Letters*, vol. 20, pp 14-16, 1984.

[Durie97] R. Durie and A. Bojanczyk, "Automated modeling of parallel algorithms for performance optimization and prediction," in *Proceedings of SIAM '97,* to be published.

[Flynn72] M. Flynn, "Some computer organizations and their effectiveness," *IEEE Transactions on Computers*, vol C-21, no. 9, pp. 948-960, Sep. 1972.

[Frigo97] M. Frigo and S. Johnson, "The Fastest Fourier Transform in the West," Massachusetts Institute of Technology, MIT Technical Report # MIT-LCS-TR-728, available at http://theory.lcs.mit.edu/~fftw.

[Gusciora96] Parallel 2-D FFT row-column source code written in C with MPI communication by George Gusciora, located at Maui High Performance Computing Center, http://www.mhpcc.edu/training/workshop/html/workshop.html.

[Heideman86] M. Heideman and C. Burrus, "On the number of multiplications necessary to compute a length-2n DFT," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 34, pp. 91-95, Feb. 1986.

[Henley97] G. Henley, et al., "BDM: A multiprotocol Myrinet control program and host application programmer interface," Mississippi State University, Technical Report #MSSU-EIRS-ERC-97-3, May 1997.

[Hennessy96] J. Hennessy and D. Patterson, *Computer Architecture A Quantitative Approach.* San Francisco: Morgan Kaufmann Publishers, 1996.

[HPCCITS96] High Performance Computing, Communications and Information Technology Subcommittee, High Performance Computing and Communications: Foundation for America's Information Future, 1996.

[Hwang93] K. Hwang, *Advanced Computer Architecture*. New York: McGraw Hill, 1993.

[IBM97] SP2 specificiations available at http://www.rs6000.ibm.com.

[Intel95] Paragon specifications formerly available at Intel website, copy available in Parallel Lab, Rm 243, Bldg 640, AFIT.

[Judd97] D. Judd et al., "Performance evaluation of large-scale parallel clustering in NOW environments," *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing*, Minneapolis, Minnesota, March 1997.

[Karamcheti94] V. Karamcheti and A. Chien, "Software overhead in messaging layers: Where Does the Time Go?," *Proceedings of ASPLOS-VI*, Oct. 1994.

[Kumar94] V. Kumar et al., *Introduction to Parallel Computing*. Redwood City, CA: The Benjamin/Cummings Publishing Company, 1994.

[Kolba77] D. Kolba and T. Parks, "A prime factor FFT algorithm using high speed convolution," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 25, pp. 281-294, Aug. 1977.

[LAM97] LAM home page - http://www.osc.edu/lam.html.

[Lamont97] G. Lamont and D. Gallagher, "Scaleable distributed multi-dimensional FFT algorithm design, implementation and analysis," Air Force Institute of Technology, Interim Report 2, Summer/Fall 1997.

[Lauria96] M. Lauria and A. Chien, "MPI-FM: High performance MPI on workstation clusters," *Journal of Distributed Computing*, vol. 40, no. 1, pp 4-18, Jan. 1997.

[MPI94] Messaging Passing Interface Forum, "The MPI message passing interface standard," Technical Report, University of Tennessee, Knoxvile, April 1994.

[Myricom97] Performance results of various platforms employing Myrinet available at http://www.myri.com.

[NAS94] D. Bailey et al., "The NAS parallel benchmarks," NASA Ames Reseach Center, RNR Technical Report #RNR-94-007, March 1994.

[NASWeb97] NAS Parallel Benchmarks home page - http://www.nas.nasa.gov

[ODonnell96] I. O'Donnell and D. Yee, "FFT implementation exploration," University of California, Berkeley, http://infopad.eecs.berkeley.edu/~ian/ee225c/report.html.

[Ooura97] T. Ooura, Dept of Applied Physics, University of Tokyo, http://momonga.t.u-tokyo.ac.jp/~ooura/fft.html.

[Pakin95] S. Pakin, et al., "High performance messaging on NOWs - Illinois Fast Messages for Myrinet," *Proceedings of Supercomputing '95*, San Diego, CA.

[Patel93] J. Patel and L. Jamieson, "Evaluating scalability of the 2-D FFT on parallel computers," *Proceedings of 1993 Computer Architectures for Machine Perception*, Nov. 1993, pp. 109-16.

[Pitas93] I. Pitas, *Digital Image Processing Algorithms*. New York: Prentice Hall International, 1993.

[Snir96] M. Snir et al., *MPI: The Complete Reference*. Cambridge, Massachusetts: MIT Press, 1996.

[Spec95] Spec CPU benchmarks, http://www.spec.org.

[Tezuka96] H. Tezuka, et al., "PM: A high performance communication library for multi-user parallel environments", Real World Computing Partnership, Japan, RWCP Technical Report #TR-96015, 1996.

[Unify97] Unify implementation available at ftp://erc.msstate.edu/unify.

[vonEicken92] T. von Eicken, et al., "Active Messages: A mechanism for integrated communication and computation," *Proceedings of the 19th ISCA*, May 1992, pp. 256-266.

# VITA

First Lieutenant David Charles Gindhart was born on September 6, 1971 in Philadelphia, Pennsylvania. Upon graduating from Strath Haven High School in Wallingford, Pennsylvania, he attended the Pennsylvania State University. He received a Bachelor of Science in Computer Engineering and was commissioned a second lieutenant in the Air Force on January 8, 1994.

David's first assignment was to Robins Air Force Base, Georgia where he was a network engineer in the Avionics Directorate. Following his tour at Robins, he arrived at the Air Force Institute of Technology.

Permanent Address: 4 Byre Lane
Wallingford PA 19086

| REPORT DOCUMENTATION PAGE | Form Approved OMB No. 0704-0188 |
|---|---|

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

| 1. AGENCY USE ONLY *(Leave blank)* | 2. REPORT DATE | 3. REPORT TYPE AND DATES COVERED |
|---|---|---|
| | December 1997 | Master's Thesis |

**4. TITLE AND SUBTITLE**
A COMPARATIVE ANALYSIS OF NETWORKS OF WORKSTATIONS AND MASSIVELY PARALLEL PROCESSORS FOR SIGNAL PROCESSING

**5. FUNDING NUMBERS**

**6. AUTHOR(S)**
David C. Gindhart, 1Lt, USAF

**7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)**
Air Force Institute of Technology
2750 P Street
WPAFB OH 45433-7765

**8. PERFORMING ORGANIZATION REPORT NUMBER**
AFIT/GCE/ENG/97D-01

**9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)**
AFRL/OCSS
Dr. Rich Linderman
26 Electronic Parkway
Rome NY 13441-4514

**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**

**11. SUPPLEMENTARY NOTES**

**12a. DISTRIBUTION AVAILABILITY STATEMENT**
Approved for public release; distribution unlimited

**12b. DISTRIBUTION CODE**
A

**13. ABSTRACT** *(Maximum 200 words)*

The traditional approach to parallel processing has been to use Massively Parallel Processors (MPPs). An alternative design is commercial-off-the-shelf (COTS) workstations connected to high-speed networks. These networks of workstations (NOWs) typically have faster processors, heterogeneous environments, and most importantly, offer a lower per node cost.

This thesis compares the performance of MPPs and NOWs for the two-dimensional fast Fourier transform (2-D FFT). Three original, high-performance, portable 2-D FFTs have been implemented: the vector-radix, row-column and pipeline. The performance of these algorithms was measured on the Intel Paragon, IBM SP2 and the AFIT NOW, which consists of 6 Sun Ultra workstations connected via the Myrinet switch.

Three important conclusions have been made. First, the pipeline was the best algorithm on all platforms by approximately 30%. Second, the NOW was nearly equal to the SP2 in runtime, while the Paragon did not outperform a single Ultra workstation. As a result, NOWs are a competitive platform for this application. Finally, only limited speedup was achieved on the SP2 (2.9) with 32 processors, and AFIT NOW (1.9) with 5 processors. It appears that the changing communication-to-computation ratio has made the 2-D FFT a less viable candidate for parallelization, given its high communication overhead.

| 14. SUBJECT TERMS | 15. NUMBER OF PAGES |
|---|---|
| Networks of Workstations, Massively Parallel Processors, Myrinet, Two-Dimensional Fast Fourier Transform, 2D FFT, Parallel Processing, Pipeline | 150 |
| | 16. PRICE CODE |

| 17. SECURITY CLASSIFICATION OF REPORT | 18. SECURITY CLASSIFICATION OF THIS PAGE | 19. SECURITY CLASSIFICATION OF ABSTRACT | 20. LIMITATION OF ABSTRACT |
|---|---|---|---|
| Unclassified | Unclassified | Unclassified | UL |

Standard Form 298 (Rev. 2-89) (EG)
Prescribed by ANSI Std. 239.18
Designed using Perform Pro, WHS/DIOR, Oct 94

## GENERAL INSTRUCTIONS FOR COMPLETING SF 298

The Report Documentation Page (RDP) is used in announcing and cataloging reports. It is important that this information be consistent with the rest of the report, particularly the cover and title page. Instructions for filling in each block of the form follow. It is important to ***stay within the lines*** to meet ***optical scanning requirements.***

**Block 1.** Agency Use Only *(Leave blank)*.

**Block 2.** Report Date. Full publication date including day, month, and year, if available (e.g. 1 Jan 88). Must cite at least the year.

**Block 3.** Type of Report and Dates Covered. State whether report is interim, final, etc. If applicable, enter inclusive report dates (e.g. 10 Jun 87 - 30 Jun 88).

**Block 4.** Title and Subtitle. A title is taken from the part of the report that provides the most meaningful and complete information. When a report is prepared in more than one volume, repeat the primary title, add volume number, and include subtitle for the specific volume. On classified documents enter the title classification in parentheses.

**Block 5.** Funding Numbers. To include contract and grant numbers; may include program element number(s), project number(s), task number(s), and work unit number(s). Use the following labels:

| | |
|---|---|
| **C** - Contract | **PR** - Project |
| **G** - Grant | **TA** - Task |
| **PE** - Program Element | **WU** - Work Unit Accession No. |

**Block 6.** Author(s). Name(s) of person(s) responsible for writing the report, performing the research, or credited with the content of the report. If editor or compiler, this should follow the name(s).

**Block 7.** Performing Organization Name(s) and Address(es). Self-explanatory.

**Block 8.** Performing Organization Report Number. Enter the unique alphanumeric report number(s) assigned by the organization performing the report.

**Block 9.** Sponsoring/Monitoring Agency Name(s) and Address(es). Self-explanatory.

**Block 10.** Sponsoring/Monitoring Agency Report Number. *(If known)*

**Block 11.** Supplementary Notes. Enter information not included elsewhere such as: Prepared in cooperation with....; Trans. of....; To be published in.... When a report is revised, include a statement whether the new report supersedes or supplements the older report.

**Block 12a.** Distribution/Availability Statement. Denotes public availability or limitations. Cite any availability to the public. Enter additional limitations or special markings in all capitals (e.g. NOFORN, REL, ITAR).

| | |
|---|---|
| **DOD** - | See DoDD 5230.24, "Distribution Statements on Technical Documents." |
| **DOE** - | See authorities. |
| **NASA** - | See Handbook NHB 2200.2. |
| **NTIS** - | Leave blank. |

**Block 12b.** Distribution Code.

| | |
|---|---|
| **DOD** - | Leave blank. |
| **DOE** - | Enter DOE distribution categories from the Standard Distribution for Unclassified Scientific and Technical Reports. |
| **NASA** - | Leave blank. |
| **NTIS** - | Leave blank. |

**Block 13.** Abstract. Include a brief *(Maximum 200 words)* factual summary of the most significant information contained in the report.

**Block 14.** Subject Terms. Keywords or phrases identifying major subjects in the report.

**Block 15.** Number of Pages. Enter the total number of pages.

**Block 16.** Price Code. Enter appropriate price code *(NTIS only)*.

**Blocks 17. - 19.** Security Classifications. Self- explanatory. Enter U.S. Security Classification in accordance with U.S. Security Regulations (i.e., UNCLASSIFIED). If form contains classified information, stamp classification on the top and bottom of the page.

**Block 20.** Limitation of Abstract. This block must be completed to assign a limitation to the abstract. Enter either UL (unlimited) or SAR (same as report). An entry in this block is necessary if the abstract is to be limited. If blank, the abstract is assumed to be unlimited.